**UNIVERSITY OF OSLO**
**Department of Informatics**

# Specialization inheritance and specialization bounded polymorphism

Marco Temperini

**Research report 226**

**ISBN 82-7368-141-6**
**ISSN 0806-3036**

**December 1996**

# Specialization inheritance and specialization bounded polymorphism

Marco Temperini*

Dip. Informatica e Sistemistica
Università "La Sapienza",
Via Salaria, 113
I-00198, Roma, Italia

and

Institutt for Informatikk
Universitetet i Oslo
P.O. Boks 1080, Blindern
N-0316, Oslo, Norge

December 1996

*email: marte@dis.uniroma1.it; Url: http://www.dis.uniroma1.it/pub/marte/

**Abstract**

We define a specialization inheritance mechanism for object-oriented programming, admitting covariant redefinition of both methods and instance variables in subclassing. We investigate on the semantic weakness that makes such very flexible inheritance infeasible for a statically type-checked programming language supporting polymorphic assignment and polymorphic method invocation.

We show that the source of troubles is not in the covariant redefinition of methods, and present a suitable multiple dispatch mechanism. This multiple dispatch uses static type information in order to drive the execution of method invocations. Moreover it exploits the notion of *method linearization* (that we define in the paper) at both compile-time and run-time.

The covariant redefinition of instance variables can be the source of run-time type errors in polymorphic instructions, when the *update problem* occurs. We devise a general mechanism to recover from the update problem. In order to be allowed to redefine covariantly class instance variables and use them in programs with polymorphism, the programmer is requested to define suitable `default-value` methods, able to produce a value for an instance variable starting from an actual value of the same instance variable in a superclass.

# Contents

# 1 Introduction

Specialization inheritance is a mechanism that allows a natural definition and treatment of objects hierarchies; still, a number of problems may arise, namely when dealing with the execution of polymorphic instructions, leading to unsafe specialization inheritance-based programs.

Here we confront such problems and discuss the treatment of polymorphism in strongly typed object-oriented programming languages.

We start from a very liberal definition of inheritance (Tab. 2); then we devise and show a permissive yet type safe management of polymorphic instructions occurring in programs based on such inheritance.

The aim is to provide the largest allowance for polymorphism, and to ensure that compile-time type correctness is not disrupted at run-time. In other words we want to avoid that `message-not-understood` errors appear during the program execution, and that badly typed assignments are executed, in consequence of polymorphic statements. We also try to model our proposals without resorting to the addition of dynamic type checks running within "type suspect" programs.

We refer to such a discipline as a *specialization bounded polymorphism*, to make clear that it is developed in a different framework than the one based on *subtyping bounded polymorphism*, that is usually advocated for this purposes ([5]).

Indeed, our definition of inheritance is so liberal that it does not ensure that the data type designed by a class is a supertype of the one defined by subclasses.

Nevertheless this kind of inheritance is supported (possibly in a limited form) by the most widely used object-oriented programming languages; so it is worth to study how far it can support safe polymorphism.

## 1.1 Specialization inheritance

Here we give a basic definition for the specialization inheritance, together with the basic notation and nomenclature.

The class construct is the usual one (see Tab. 1): it features a set of *instance variables*, and a set of *methods* (functions operating over the instance variables). We will call generically both instance variables and methods as *attributes* of the class.

Each instance produced starting by a given class (an *object of that class*), will maintain a *state*, represented by the instance variables concretely stored in it, and will provide a behavior, through the methods defined in the class.

By {inh-list}, the list of classes that are *inherited* by the new class is provided. If this list has a single element we say that the new class is produced by *single inheritance*. If there are several classes inherited, we use the term *multiple inheritance*.

Once some classes are declared in a program, also variables can be declared, as references to objects of a given class: `c:`$C$. Actually this declaration can be interpreted in two very different ways: `c` could be a *value*, representing the object

| |
|---|
| class *Class_Name* {inh-list} |
| $\mathtt{v}_1 : C_1$ ; // instance variables |
| ... |
| $\mathtt{v}_m : C_m$ ; |
| |
| $\mathtt{meth}_1(\mathtt{p}_1 : P_1) : R_1$ is {$\mathtt{m}_1$ body} // methods definitions |
| ... |
| $\mathtt{meth}_n(\mathtt{p}_n : P_n) : R_n$ is {$\mathtt{m}_n$ body} |
| endclass |

Table 1: Class construct

itself, or a *pointer* to such a value. Usually, supporting values is simpler than handling pointers[1]. In the sequel we will always refer to pointers and everything that will be stated for pointer variables could be stated as well for plain values. So `c` will denote a pointer to objects of class $C$. Once such an object is referred to by `c`, its attributes are accessed by the *dot-notation*, `c.v` standing for the access to the attribute `v` of the object referred to by `c`[2]. When the attribute is a method, as in `c.m(...)`, the access is called *method invocation*: a function call addressed to an object, that means to ask the object for the execution of one of its behaviors.

To indicate "the" method `m` defined in class $C$, we will use sometimes the notation $C$`::m`, as in C++ ([19]). Without loss of generality, we will deal with methods with a single parameter, and adopt the notation `m(`$P$`):`$R$ to mean that $P$ is the class of the argument and $R$ is the class of the result.

If the class $\underline{C}$ inherits $\overline{C}$ we say, as usual, that $\underline{C}$ is a *subclass* of $\overline{C}$ (write $\underline{C} \leq \overline{C}$), and that $\overline{C}$ in turn is a *superclass* of $\underline{C}$.

The inheritance relation is here the *specialization inheritance* that we define as *strict inheritance* with *covariant redefinition* of attributes. By *strict inheritance* we mean that all the attributes of $\overline{C}$ are inherited in $\underline{C}$ and they are part of the $\underline{C}$ definition. By *redefinition* we mean that the inherited attributes can be redefined in the subclass: the usual effect of an attribute redefinition is the so called *overriding* of the previous definition of the attribute. The redefinition of an attribute must be *covariant*. The rules of covariant redefinition are given through the definition of our specialization inheritance, included in Tab. 2.

Roughly, the rules for covariant redefinition state that a redefinition must involve subclasses of those involved in the previous definition.

Some covariant definitions are shown in Fig. 1: the instance variable `rank` is rede-

[1]Basically this is because, given two types $\mathcal{T}$, $\mathcal{T}'$, $\mathcal{T}'$ subtype of $\mathcal{T}$, the related pointer types are no more in the subtyping relation.

[2]Note that to allow free access to an object instance variable from outside the object is not very advisable, in fact, due to the violation of encapsulation principles. Here we are dealing with other aspects of object-oriented programming and don't care about this.

**single inheritance** Let be $\underline{C} \leq \overline{C}$;

1. Let $\mathtt{v}{:}V_{\overline{C}}$ be an instance variable declared in $\overline{C}$. The definition $\mathtt{v}{:}V_{\underline{C}}$ in $\underline{C}$ is then a covariant redefinition if $V_{\underline{C}} \leq V_{\overline{C}}$.
2. Let $\mathtt{m}(P_{\overline{C}}){:}R_{\overline{C}}$ be a method declared in $\overline{C}$. The definition $\mathtt{m}(P_{\underline{C}}){:}R_{\underline{C}}$ in $\underline{C}$ is then a covariant redefinition if $P_{\underline{C}} \leq P_{\overline{C}}$ and $R_{\underline{C}} \leq R_{\overline{C}}$.

**multiple inheritance** Let be $\underline{C} \leq \overline{C}_1, \ldots, \overline{C}_n$;

1. if $\mathtt{v}{:}V_{C_k}$, in $C_k$, is the only such definition given in the superclasses, then $\mathtt{v}{:}V_{\underline{C}}$, in $\underline{C}$ is a covariant redefinition if $V_{\underline{C}} \leq V_{C_k}$.
2. if there are several (re)definitions of a homonymous instance variable in superclasses ($\mathtt{v}{:}V_{C_{k_1}}$, ..., $\mathtt{v}{:}V_{C_{k_m}}$ with $[k_1, \ldots, k_m] \subseteq [1, \ldots, n]$:), then $\mathtt{v}{:}V_{\underline{C}}$, in $\underline{C}$ is a covariant redefinition if $\forall i \in [k_1, \ldots, k_m]$ is $V_{\underline{C}} \leq V_{C_{k_i}}$.
3. if there are several (possibly one) definitions of a homonymous method in the superclasses ($\mathtt{m}(P_{C_{k_1}}){:}R_{C_{k_1}}, \cdots, \mathtt{m}(P_{C_{k_1}}){:}R_{C_{k_m}}$, with $[k_1, \ldots, k_m] \subset [1, \ldots, n]$:), then $\mathtt{m}(P_{\underline{C}}){:}R_{\underline{C}}$ in $\underline{C}$ is a covariant redefinition if $P_{\underline{C}} \leq P_{C_{k_i}}$, and $R_{\underline{C}} \leq R_{C_{k_i}}$, forall $i \in [k_1, \ldots, k_m]$.

The definition of a class $\underline{C}$ is *legal* whenever any redefinition follows the above rules, with the additional constraint that, in case (2) of multiple inheritance the covariant redefinition of v:$V_{\underline{C}}$ is mandatory if there isn't already a *minimal* instance variable (a $\mathtt{v}{:}C_{k_i}$ such that $\forall j \in [k_1, \ldots, k_m]$ is $V_{C_i} \leq V_{C_j}$).

Table 2: Enhanced Strict Inheritance

fined in the class *Special_Student* (assuming *Ext_Evaluation* $\leq$ *Evaluation*), and the method `TeachPersonally` (that has no return type) is redefined in *Special_Teacher*.

Our interpretation of these cases is that the objects of class *Special_Student* will feature a single `rank` instance variable of class *Ext_Evaluation* and that the objects of class *Special_Teacher* will show *in primis* the specialized behavior *Special_Teacher*::`TeachPersonally`, and possibly the *Teacher*::`TeachPersonally`, when needed[3].

To make the notation less cumbersome, we don't embed in this definition the distinction between *private* and *public* attributes. It is not the main concern in the scope of this paper, so we will consider the whole set of attribute definitions as the *interface* of our classes.

Last notion to enlist is the *applicability* of a method definition. Assume to have the method invocation `o.m(q)`, where `q` denotes an object of class $Q$, then a method `m(`$P$`)`:$R$ is *applicable* to the method invocation iff $Q$ is a subclass of $R$.

Of course our inheritance doesn't entail the usual notion of subtyping ([2, 5], But there are many cases in which classes defined by the enhanced strict inheritance can be used in programs with polymorphism. What we are going to do in the following is to investigate on the semantic weaknesses of such a permissive specialization inheritance, providing some solutions to let it be used in a statically type checked programming language and enjoy its high expressive power.

### 1.2 Troubles from "specialization bounded" polymorphism

In Sec. 2 the problems found when using polymorphism in the sample system of Fig. 1 are discussed. They arise, for example in the program excerpts of Fig. 2, and are of a twofold nature.

First, there is a strictly *behavioral* problem, due to covariant redefinition of methods: given a method invocation `t.TeachPersonally(s)` (`t`: *Teacher*, `s`: *Student*), which is statically correct, what method should be executed at run-time once `t` refers to an object of *Special_Teacher* and `s` to an object of class *Student*? A solution based on *multiple dispatch* by *abstraction level* is presented in Sec. 3.

Second, there is a *structural* problem, due to the liberal use of covariant redefinition of mutable instance variables (data members on which an update might occur). A solution based on the definition of additional *default-value methods* is presented in Sec. 4. This solution, 1) fits well in the method discipline we have defined, since it is based on the addition of new methods that should "help" instance variables assignments, and that are managed as normal methods; but 2) cannot always be applied: it is applicable in all cases where the replacement of an object of superclass by another of subclass does make sense based on some design insight.

[3]We interpret *Teacher*::`TeachPersonally` as a "behavioral heritage" that could be used when a special teacher is asked to act in the environment where a normal teacher is expected.

# 2 Specialization bounded polymorphism

In Fig. 1 class hierarchies are presented, where both variable and method redefinitions appear. In this section we discuss about how much polymorphism is likely to be supported, and point out what kinds of troubles it can produce once used in programming over specialization inheritance class hierarchies.

Looking at the figure, we can state the following analysis for our "toy-school".

## 2.1 Analysis

A student must take several courses. Normal students take only normal courses. A curriculum cannot be composed of only special courses (they are too few), so special students will have to take both special and normal courses.

So, from an "objects" viewpoint, we need that in any moment an instance of *Special_Student* can appear where a plain *Student* is expected. Moreover our model supports that a normal teacher (teaching a normal course) teaches special students (taking that normal course).

Special teachers teach special courses, but there are also "emergency cases" when they must teach normal courses. This means that for the teachers, the same polymorphism we noticed for the students holds: the possibility of having an instance of *Special_Teacher* where one of *Teacher* is expected must be supported.

Then, some special teacher could have to deal with both normal students and special ones (if s/he teaches a normal course). We will analyze the behavior of such a teacher in Sec. 2.2. This behavior is represented by the methods `TeachPersonally` and `Interrogate?`, which give the interactions with a single student (in different respects, also from a type safeness viewpoint).

## 2.2 Problems with specialization bounded polymorphism

We limit our discussion to imperative programming languages. So we assume to have to deal with notions such as pointer variables and assignment operator(s). In order to express the above polymorphic effects, we allow a program to feature

1. the use of variables: we also state that they must be declared such as in

   `s:` *Student*; `t:` *Teacher*; `ss:` *Special_Student*; `st:` *Special_Teacher*;

2. that such variables can be heterogeneous, i.e. they can be polymorphically assigned as in (besides the obvious `s := new` *Student*;, `t := new` *Teacher*;)

   `s := new` *Special_Student*; or `t := new` *Special_Teacher*;

3. that a variable cannot be assigned to an instance of a superclass of the expected class (`ss := new` *Student* is illegal)

```
class Student { Person}
psycho_char: string              // psychological characterization
rank: Evaluation                 // Evaluation = {mark:integer}
Set_rank(p: Evaluation)   is {rank := p; };
endclass

class Special_Student { Student}
rank: Ext_Evaluation             // Ext_Evaluation = {mark:integer; ability:string}
Set_rank(p: Ext_Evaluation)   is {rank := p; };
endclass

class Teacher { }
TeachPersonally(p: Student) is {··· uses psycho_char and rank ···};
Interrogate?(p: Student)
  is {e:=new(Evaluation); ··· p.Set_rank(e); ··· };
endclass

class Special_Teacher { Teacher}
TeachPersonally(p: Special_Student) is {··· uses psycho_char and rank ···};
Interrogate?(p: Special_Student)
  is {e:=new(Ext_Evaluation); ··· p.Set_rank(e); ··· };
endclass

                    class School {}
                    T: list_of Teacher;
                    S: list_of Student;
                    C: list_of Course;
                    SC: list_of Special_Course;
                    endclass

class Course { }
aTeacher:Teacher;
aClass: list of Student;
Days:string;
endclass

                                  class Special_Course { Course}
                                  aTeacher:Special_Teacher;
                                  aClass: list of Special_Student;
                                  endclass
```

Figure 1: a polymorphic school

```
s := new Student
t := new Special_Teacher
t.TeachPersonally(s)
```
**(a)**
(special teacher teaching normal student)

```
s := new Special_Student
t := new Teacher
t.Interrogate?(s)
```
**(b)**
(normal teacher interrogates special student)

Figure 2: error prone polymorphic statements

4. that a *polymorphic method invocation*[4] such as `t.TeachPersonally(s)` is statically type checked, in order to state its compile-time legality and foresee its run-time safeness.

   In fact the above invocation could have several meanings at run-time (depending on the *run-time class* of `t` and `s`, i.e. the class of the object that they refer to, at the moment of execution). If we point out by $\mathtt{v}^{Actual}$ that the variable `v` at run-time refers to an object of exact class *Actual*, we have the following possible cases:

$$\mathtt{t}^{Teacher}\mathtt{.TeachPersonally(s}^{Student}\mathtt{)}$$
$$\mathtt{t}^{Special\_Teacher}\mathtt{.TeachPersonally(s}^{Special\_Student}\mathtt{)}$$
$$\mathtt{t}^{Teacher}\mathtt{.TeachPersonally(s}^{Special\_Student}\mathtt{)}$$
$$\mathtt{t}^{Special\_Teacher}\mathtt{.TeachPersonally(s}^{Student}\mathtt{)}$$

In our framework the set of instructions in Fig. 2 would give severe type errors at execution. Let us discuss the two cases, separately.

### 2.2.1 case (a): covariant method redefinition

Here we deal with the notion of what is the behavior of a teacher when s/he is given a normal student to teach to. That is, what method must be selected to execute the last instruction in Fig. 2(a). In this respect, different languages may operate in different ways. The most common method selection algorithm, would here execute *Special_Teacher*::`TeachPersonally`. And it would fail since a normal student were passed as argument where at least a special one is expected.

A solution to this problem is a matter of giving a suitable semantics to the inheritance and to the method selection. In particular, a normal teacher could teach a special student, in this case there are no special abilities of the student to be exploited and the behavior of the teacher should conform to the one previously defined in *Teacher*.

[4](an invocation where the receiver's and argument's actual classes could be subclasses of the static ones)

In other words, if the special teacher is requested to operate with a special student s/he can perform his/her peculiar *Special_Teacher*::`TeachPersonally`. On the other hand, *if that method is not applicable* there is *Teacher*::`TeachPersonally` available, i.e. the method that *Special_Teacher* inherited but redefined.

By the way, this subsidiary method was the one that did let the static type checking algorithm declare correct our sample method invocation. So, it makes sense that we eventually use this method, when all the others, more specialized, fail.

Differently than in other approaches ([3, 9]), we think that this policy can be adopted only for statically correct polymorphic method invocations. By such method invocations we mean those whose receiver has a static class that is enough to perform the requested behavior. For example we think that `st.TeachPersonally(s)` shouldn't be declared statically correct. Otherwise we would allow a special teacher to perform a *Teacher* behavior, in a context where only special teachers (or more specialized objects) are expected as receiver. If *Special_Teacher* has overridden that behavior we should consider it in the static contexts where no less than special teachers are expected. And there is no method `TeachPersonally` in *Special_Teacher*, which is applicable to a normal student.

So the point is that a special teacher can be seen *as* a normal teacher, but only "for emergency", i.e. when this occurs in a polymorphic instruction where plain teachers where statically expected.

We let static information be considered for method invocations by the notion of *abstraction level*.

Given `o.m(q)`, if in the static class inferred for the receiver expression `o` there is an applicable method, that class is the abstraction level of `o.m(q)`. If there is no explicit definition of `m` in the static class of `o`, we consider the most recent inherited definition (or definitions, in case of multiple inheritance). If it is applicable we have the abstraction level, otherwise the method invocation is illegal. Once an abstraction level is stated for `o.m(q)`, of course `o` remains of its own class; but if its own behavior is not correct w.r.t. the run-time context of the invocation, `o` can eventually switch to inherited behaviors ("restore the ancestral characters").

The abstraction level of the special teacher referred to by `t` in the example above is *Teacher*, and this means that `t` can consider also *Teacher*::`TeachPersonally` among its possible behaviors, if needed. The abstraction level of the special teacher referred to by `st` is *Special_Teacher*; so it knows from compile-time that there is only *Special_Teacher*::`TeachPersonally` available.

Sec. 3 provides a complete discussion about this semantics of inheritance and method lookup. There, we apply the *multimethod* approach to our imperative setting, taking care of the abstraction level and of its application to a multiple inheritance environment.

### 2.2.2 case (b): covariant instance variable redefinition

In the example of Fig. 2(b) the method *Teacher*::`Interrogate?`(`p`:*Student*) is polymorphically executed with a special student as argument. Then, a run-time type error occurs when the *updating method*
*Special_Student*::`Set_rank`(`p`:*Ext_Evaluation*) is called to execute with an argument of class *Evaluation*. This is an *update problem*: it is the reason why the specialization bounded polymorphism is unsafe and the subtyping relation excludes covariant redefinition of data members.

In fact, problems with method redefinition occur only in such cases. If there were no instance variable redefinitions in our example, covariant method redefinition with multiple dispatch would be enough (for case 2(a)) and safe. If we excluded data member specialization and limited the specialization inheritance to only support covariant method redefinition, we can manage a sound specialization bounded polymorphism. But, this severely weakens the expressive power of a language.

In Sec. 4 we show how instance variable redefinitions and instructions like `t.Interrogate?(p)` can be allowed in a program, still preserving that their static correctness is followed by run-time correctness.

Our technique consists of coupling each covariant redefinition of a data member with an "ad hoc" *default-value* method that is activated when an apparently unsafe assignment is going to be performed (an instance variable of superclass being assigned by a value of subclass). This method must be able to produce a value for an instance variable, starting from an object the variable could have if not redefined.

Basically, we modify the semantics of the assignment operator used within an updating method, so that, when the object at the right hand side isn't in accord with the left-hand side it might call a default-value method.

The default-value methods are treated as normal methods and their execution follows the semantics of our method lookup.

# 3 Multiple dispatch via abstraction level and method linearization

Given a method invocation `o.m(q)`, the static type checking considers the class inferred for the expressions `o` and `q` (their static classes) and checks whether there is a method available in the static class of the receiver `o` that can be applied with argument of the static class of `q`. In addition, it checks that such an applicable method returns a result (if any) coherent with the context of the method invocation. So, when a method invocation is declared "legal" by the static type checking, it is supposed to be ensured that there will be an executable method at run-time and that its result will be coherent with the expected one (the returned object's class will be subclass of the class inferred at compile time). Such a method *makes the invocation statically correct*.

We have seen in Sec. 2.2.1 that the redefining methods, while by definition return

a coherent result, could be non-applicable (non-executable) at run-time. And we have sketched the solution as follows. We assume that the run-time class of an expression must be a subclass of the static one: then the method that *makes the invocation statically correct*) is polymorphically applicable at run-time and produces a result of subclass of the expected one. So at least this method can be executed to satisfy the method invocation. However, if between the actual and the static class there is a class with a better (*more specialized*) definition of the method, and if it is applicable, then that is the method that will be selected for execution.

In case of single inheritance this policy can be applied with no further discussion: given a statically correct method invocation `o.m(q)`, to be executed, the following *lookup* function returns the most specialized applicable method found in the search space $\mathcal{S}^{\underline{O},\overline{O}} = \{C \text{ such that } \underline{O} \leq C \leq \overline{O}\}$, where $\overline{O}$ is the abstraction level of `o.m(q)` and $\underline{O}$ is the actual class of `o`. Note that in this recursive function $\mathcal{S}^{\underline{O},\overline{O}}$ is passed for $\mathcal{N}$ at the first call.

**Definition 1 (*lookup*, with abstraction level: single inheritance)**

*lookup* (o, m, q, $\mathcal{N}$) =
  **if** m is defined in *car*($\mathcal{N}$)
    **then**
      **if** m is applicable with argument q
        **then return** (*car*($\mathcal{N}$)::m)
        **else return** ( *lookup* (o,m,q, *cdr*($\mathcal{N}$)))
**end** *lookup*

□

This is a straightforward application of the multiple dispatch advocated by the usual multimethod approach. Note that only a portion of the *generic function* ([3]) m is explored, limited to the subset $\mathcal{S}^{\underline{O},\overline{O}}$ of the "cone" of the superclasses of $\underline{O}$ that define m.

Things become different in the case of multiple inheritance. Then, the inheritance hierarchy (or $\mathcal{S}^{\underline{O},\overline{O}}$) is a directed acyclic graph on which the inheritance relation defines a partial order.

Let us suppose that `o.m(q)` must be executed, where $\overline{O} = B$ (i.e. o is an expression returning an object of static class $B$) and $\underline{O} = D'$ (the o expression actually returns an object of class $D'$). Refer to Fig. 3, and suppose that $D'$::m isn't applicable, while all the others are so. After that $D'$::m has been seen non-applicable, what method (class) should be considered at next? This is the so-called *conflict resolution problem*

There are several proposals to solve conflicts. For an account of them see App. B.1. Basically these proposals can be divided into two families: one family collects solutions that avoid conflicts by means of classes modifications (renaming, forced redefinition of a *least* method). The second family advocates the use of a *linearization* of the graph $\mathcal{S}^{\underline{O},\overline{O}}$. This linearization is a *linear extension* ([13]) of the above graph, and can be used to perform a sequential method lookup. The

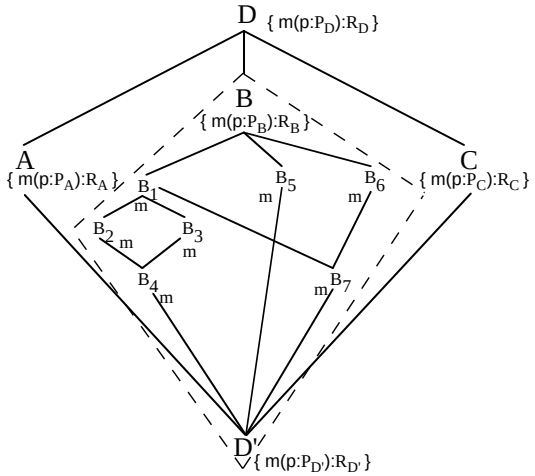



Figure 3: multiple inheritance dag

solution we propose is in the latter family (and is devised to fix some drawbacks of the others in the same group).

Given a class, there are several techniques for constructing a linearization of the *cone* of its superclasses. There are *depth-first* and *breadth-first* techniques. Some classical depth-first strategies are analyzed are discussed in [13]; one of these techniques is applied in the *Common Lisp Object System* (CLOS) [3]; in [10, 11, 12] a breadth-first strategy is adopted. In this paper we don't care what linearization algorithm is chosen. Whatever it might be, we call *class linearization* or *C-linearization* the chain of the superclasses of a class $C$. However, the general properties of a linearization algorithm, on which we rely, are i) that it associates to each class $C$ a unique $C$-linearization; ii) that the $C$-linearization is stated on the basis of the inheritance order of $C$ (so the *inheritance order* in which the direct superclasses appear in {inh-list} is significant); and iii) that in a linearization a class never precedes one of its subclasses. The following is a breadth-first based $D'$-linearization: $\{D', A, B_4, B_5, B_7, C, B_2, B_3, B_6, B_1, B, D\}$.

### 3.1 Pure multiple dispatch

Usually, multimethod languages don't deal with class linearizations; rather they manage a single general linearization of all the classes involved in a program. Moreover, usually the methods are not defined directly into classes, but they are declared apart, as functions. The first argument of these functions is interpreted as the owner class: sending a method invocation to an object, imply a method call with that object as first argument. Namely, all the `m`'s defined in the program are collected in a list of functions, (the *generic function*), which is ordered by a linearization al-

gorithm applied to the arguments classes (so the ordering over the first argument class is prevailing in that linearization).

The method invocations are function calls in which the first argument plays the role of the object receiving the method invocation. So the executable method is selected by looking "up" in the generic function. The first parameter to be checked is the one indicating the receiver object. So the first attempt of retrieving the method is done in the run-time class of the receiver. Looking up in the generic function means to "visit" all the other functions having such a run-time class as first parameter or a greater one.

If we "see" this behavior projected on class linearizations, we have that, the above method invocation `o.m(q)` is executed by a method lookup in the whole $D'$-linearization. This involves to check the applicability of all the methods in the cited cone, in the linearization order, until one is found applicable. So also $A$`::m` could be selected, i.e. a method that is defined in a class which is unrelated to the $B$ static class of the `o` expression. This comes out to be a drawback.

For a deeper analysis of such a drawback, see App. B.1. Just as an intuition we stress that at run-time we are going to fetch a behavior from a class that is unrelated to the one that was used for the static type checking. Indeed, at static time we "approved" `o.m(q)` since the static information allowed to verify that a method could have been found applicable at run-time. On the other hand, we also inferred a class for the result produced by the method invocation (e.g. we stated that $B$`::m` made the invocation correct, and returns an object of class $R_B$). This information could have been used during further type checking, so its validity *must* be maintained at run-time. But now, at run-time, `o.m(q)` returns an object of class $R_A$ which is potentially incompatible with the expected $R_B$.

Making the multiple dispatch *sensitive to the static context* of the method invocation (i.e. letting it use the abstraction level) provides a solution to this problem.

### 3.2 Context sensitive multiple dispatch

The search space for the method lookup algorithm, in the above pure multiple dispatch approach, is the whole cone of the superclasses of the receiver's run-time class. Looking again at Fig. 3, such redundant search space is $\mathcal{S}^{D',D}$. By using the abstraction level of the receiver expression we can bound the search space so to contain only "significant and correct" methods. For example, in Fig. 3 the class $D$ could be excluded by the search space (its `m` is "covered" by the surely applicable $B$`::m`). More important, $A$ and $C$ should be excluded for their `m` definitions aren't applicable.

In our method lookup the search space for the example of Fig. 3 would be the "diamond" $\mathcal{S}^{D',B}$, pointed out by dashed lines.

We define this method lookup just as an application of the function *lookup* over a different search space: Denote by $\mathcal{S}^{\underline{O},\overline{O}} \overset{\mathcal{L}}{\cap} \underline{O}$*-linearization* the *ordered intersection* of the operated sets, as the list of all classes that are both in $\mathcal{S}^{\underline{O},\overline{O}}$ and $\underline{O}$*-linearization*, taken in the order given by the $\underline{O}$*-linearization*.

**Definition 2 (*lookup*$^d$, with abstraction level: single diamond)**

*Given a statically correct method invocation* `o.m(q)` *with abstraction level* $\overline{O}$ *and actual class of* `o` $\underline{O}$

lookup$^d$ *(*`o`*,* `m`*,* `q`*,* $\underline{O}$*,* $\overline{O}$*)* = lookup *(*`o`*,* `m`*,* `q`*,* $\mathcal{S}^{\underline{O},\overline{O}} \overset{\mathcal{L}}{\cap} \underline{O}$-linearization*)* □

This method lookup guarantees that (1) an executable method is found at run-time (at most in the upper bound of the search space, i.e. the abstraction level: $B$); and (2) that the result of the selected method is of subclass of the expected one, since the research path is built so that it doesn't contain classes that are unrelated to the statically expected one (all methods from $B$ downward in the hierarchy (could) define methods that are covariant specializations of the previous ones; so the result is getting lower and lower but still consistent with $R_B$).

In the examples on which we based our discussion all the classes contained a definition of the invoked method. This is not always the case, of course. In the general case we admit that only the effective compile-time class of the receiver object, or one of its superclasses where the method was defined latest, can make the invocation statically correct. Such a complicate definition is needed to avoid that a receiver object expression is type checked statically correct (legal) while even at compile time it has to rely on some overridden behaviors. In this case the receiver expression used by the programmer is far too specialized for the behavior that is requested. So the programmer should be requested to provide a more general expression for that use in that context to be statically correct. This restriction doesn't appear in the usual statically typed approaches to multimethods. In our opinion it is important for making the interpretation of a program coherent with the programmer's intentions. Even at cost of bothering the programmer him/herself.

The next definition conveys our interpretation of the safety conditions for a method invocation.

**Definition 3 (static correctness of a method invocation `o.m(q)`)**

Let `o.m(q)` be a method invocation to be statically type checked; let $O$ be the static class of the expression `o`; let $Q$ be the static class of the expression `q`.

If $O$ contains a definition `m(`$P_O$`)`$:R_O$:

> if $O$`::m` is applicable to `o.m(q)`, then the invocation is statically correct and the abstraction level is $O$.

If $O$ doesn't contain a definition of `m`,

> let $\mathcal{M}$ be the set of superclasses of $O$ that contain a definition of `m`; let $\mathcal{M}'$ be the subset of $\mathcal{M}$ such that no class in $\mathcal{M}'$ has a subclass in $\mathcal{M}'$ itself:
>
> if $\exists C \in \mathcal{M}'$, such that $C$`::m` is applicable to `o.m(q)`, then the invocation is statically correct and the abstraction level is $C$.

in all other cases the invocation is not statically correct.

□

(From an "operational" viewpoint, we assume that $\mathcal{M}'$ is scanned in the $O$-linearization order, while looking for an applicable $C$::m.)

So the general aspect of the search space of our method lookup is as in Fig. 4, where the abstraction level could not coincide with the static class of the receiver object. We denote the set of classes bounded by the dashed lines in figure as $\mathcal{S}^{\underline{O},O,\overline{O}}$.

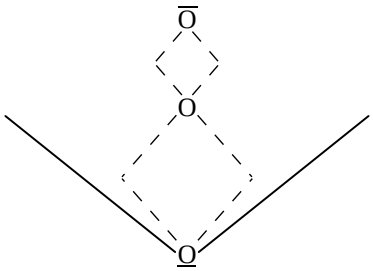



Figure 4: double diamond

We define the method lookup $lookup^{dd}$, again as an application of the function *lookup*, where the search space must be a double diamond[5], considered in the $O$-*linearization* order.

**Definition 4 ($lookup^{dd}$, with abstraction level: double diamond)**
Given a statically correct method invocation `o.m(q)` with abstraction level $\overline{O}$, static class of `o` $O$ and actual class of `o` $\underline{O}$,

$lookup^{dd}$ (o, m, q, $\underline{O}$, O, $\overline{O}$) = *lookup* (o, m, q, $\mathcal{S}^{\underline{O},O,\overline{O}} \overset{\mathcal{L}}{\cap}$ $\underline{O}$-*linearization*) □

The sketched method lookup algorithm provides the programmer with an "abstraction level driven" multiple dispatch; it exploits both the static information provided by type-checking and the dynamic information on actual objects occurring in the method invocation expression. The achievements are as follows.

- We can use the inheritance of Tab. 2 without any constraints on class definition and method redefinition (but the covariance).
- Once a method invocation has been statically type checked (as legal), there is a method executable at run-time. In the worst (topmost) case it will be selected at the abstraction level, so the receiver never behaves more generally than it was expected at compile-time.

[5]possibly degenerating in a single one, if the static class of the receiver and the abstraction level coincide

- There is no result compatibility problem, since the method dispatched by *lookup*$^{dd}$ produces a result coherent with the static and dynamic context of the invocation.

**Proposition 1 (property of *lookup*$^{dd}$)**
Given a statically correct method invocation `o.m(q)`. Let $\overline{O}$ be the abstraction level, and $O$ the static class of the receiver. Let $Q$ be the static class of `q` and $R$ the result static class. Let $\underline{Q} \leq O$ be the actual class of `o`, and $\underline{Q}$ the actual class of `q` when the method invocation is executed.

Then there is at least one method executable at run-time. Moreover, whatever the method actually selected for execution is, it returns an object of class $R' \leq R$.

***Proof***

Since the invocation is statically correct, the method $\overline{O}$`::m(p:`$P_{\overline{O}}$`):`$R_{\overline{O}}$ is statically (so dynamically) applicable. Hence, in the hypothesis it is $R = R_{\overline{O}}$.

Define $\mathcal{S} = \mathcal{S}^{\underline{Q},O,\overline{O}} \overset{\mathcal{L}}{\cap} \underline{Q}$*-linearization*. Then,

1. $\mathcal{S} \neq \emptyset$, since at least $\overline{O} \in \mathcal{S}$, and $\overline{O}$`::m` is defined.
2. $\forall C \in \mathcal{S}$, it is $\underline{Q} \leq C \leq \overline{O}$. So $\forall C \in \mathcal{S}$, the result of $C$`::m` is of class $R_C \leq R$, by the covariant rule for method redefinition.

To execute `o`$^{\underline{Q}}$`.m(q`$^{\underline{Q}}$`)`, *lookup*$^{dd}$ (`o.m(q)`, $\mathcal{S}^{\underline{Q},O,\overline{O}} \overset{\mathcal{L}}{\cap} \underline{Q}$*-linearization*) is activated. It finds at least $\overline{O}$`::m`. Moreover, from 2. we have that any other method that could be selected at run-time returns a result of class $R' \leq R$.

△

Yet, there are more problems we should confront, and we will discuss (and try to solve) in the following part of this section:

1. Our solution, at the moment, doesn't solve the conflicts but using a class linearization order for the multiple dispatch. The first applicable method encountered along the linearization is selected. As a matter of facts it is ensured type-correct; its selection would be predictable just by looking at the inheritance graph; but it must be admitted that in such a selection policy there is some unclear semantical aspects. As far as we know, this is the main argument used to support other approaches, against the multiple dispatch over linearizations. See App. B.2 for further discussion.
2. Moreover, in our case this problem has a reflex at compile-time, when the abstraction level of a method invocation is also selected on the basis of a class linearization.
3. Finally, according to multiple dispatch, there is no guarantee that the method selected after looking up along the class linearization is "the most specialized" among its companions. This means that, once a method was selected at

run-time, there is the possibility that some other methods, further in the linearization, were strictly more specialized than that.

Problems 1 and 3 are common to all the known multiple dispatch based approaches. The problem 2 is more connected to ours.

The next subsection presents the concept of *method linearization*, that helps in solving the above problems. The resulting method lookup algorithm, $lookup^m$, will still allow to select a behavior coherently with the static environment of the method invocation, and will ensure that none of the unselected methods would have been more specialized.

### 3.3 Method linearization

For languages that support multiple inheritance, the conflicts are either forbidden, as in C++ ([19]), or Eiffel ([16]), or allowed with some constraints, as in the functional schema of [9]. Otherwise they are allowed, but solved via a "first found is selected" policy, as in CLOS ([3]) or in our approach in [10].

The linearization-based selection strategy is predictable once the programmer knows its details, but shows casual aspects that weaken its appeal. For example, in [4] it has been banished, stating that if there are conflicting methods in the search space, the method lookup algorithm should jump over them till the first suitable common superclass (and related applicable method). We think that this solution wraps method specializations and makes them invisible: the trend is to apply the most general methods more widely than it is necessary, while the attitude should be the opposite.

We propose a different solution: instead of using always the same class linearization for looking up the methods, we let each method in a class be coupled to a related class ordering, that should be used for the method lookup, and that we call *method linearization*. So, when the method lookup is running between $\underline{O}$ and $\overline{O}$, for executing `o.meth(q)`, the searching order is not given by the $\underline{O}$-linearization, but by the $\mathcal{M}^{\underline{O}}_{\texttt{meth}}$ method linearization. $\mathcal{M}^{\underline{O}}_{\texttt{meth}}$ is a sequence of the superclasses of $\underline{O}$ where `meth` is explicitly defined. It is built by following the $\underline{O}$-linearization, such that never a class precedes another one where a more specialized `meth` is defined.

Our first observation is that, given two methods, it is possible to make some analysis, to understand whether one is more/less specialized of the other or they are incomparable. The analysis can be done by comparing the argument and result classes of the methods. $\mathcal{M}^{\underline{O}}_{::\texttt{meth}}$ is the result of such an analysis, carried on over the (methods defined in the) classes of the $\underline{O}$-linearization.

So each class has peculiar linearizations to submit to the method lookup algorithm, one for each method it defines.

In the following discussion we denote by $\texttt{m} <_{\mathcal{M}} \texttt{m}'$, $\texttt{m} >_{\mathcal{M}} \texttt{m}'$, $\texttt{m} \not<_{\mathcal{M}} \texttt{m}'$, that `m` is, respectively *more specialized*, *less specialized*, or *incomparable* w.r.t. `m'`. By $C < C'$, $C > C'$, $C = C'$, $C \not< C'$, we denote that $C$ is, respectively a (strict) subclass of

$C'$, a (strict) superclass of it, the same class or that $C$ and $C'$ are unrelated. The relationships that can occur between two methods are represented in Tab. 3.

| $\mathtt{m}(P):R \sim \mathtt{m}'(P'):R'$ | $R < R'$ | $R = R'$ | $R > R'$ | $R \not\sim R'$ |
|---|---|---|---|---|
| $P < P'$ | $<_{\mathcal{M}}$ | $<_{\mathcal{M}}$ | $>_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ |
| $P = P'$ | $<_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ | $>_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ |
| $P > P'$ | $<_{\mathcal{M}}$ | $>_{\mathcal{M}}$ | $>_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ |
| $P \not\sim P'$ | $<_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ | $>_{\mathcal{M}}$ | $\not\sim_{\mathcal{M}}$ |

Table 3: method specialization

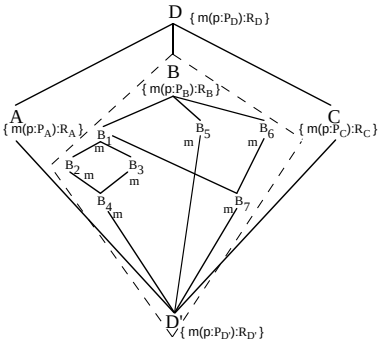
If $\{P < P',\ R < R'\}$, we can say $\mathtt{m}(P):R <_{\mathcal{M}} \mathtt{m}'(P'):R'$. We can extend this conclusion also to the cases $\{P = P',\ R < R'\}$ and $\{P < P',\ R = R'\}$. If either $\{P > P',\ R > R'\}$ or $\{P > P',\ R = R'\}$ or $\{P = P',\ R > R'\}$ we infer $\mathtt{m}(P):R >_{\mathcal{M}} \mathtt{m}'(P'):R'$. If $\{P = P',\ R = R'\}$ we infer $\mathtt{m}(P):R \not\sim_{\mathcal{M}} \mathtt{m}'(P'):R'$. For the inner part of the table, there remain only the cases $\{P < P',\ R > R'\}$ and $\{P > P',\ R < R'\}$[6]. Our attitude is to give priority to the result specialization, so we state $\mathtt{m}(P):R >_{\mathcal{M}} \mathtt{m}'(P'):R'$ for $\{P < P',\ R > R'\}$ and $\mathtt{m}(P):R <_{\mathcal{M}} \mathtt{m}'(P'):R'$ for $\{P > P',\ R < R'\}$. Last column and row deal with cases of incomparable classes. In these cases we again privilege the result class relationships. If the result classes cannot help in deciding, we state the incomparability of the methods.

Some choices in Tab. 3 need some discussion. In the cell $\{P = P, R{=}R\}$ we could set $\mathtt{m} =_{\mathcal{M}} \mathtt{m}'$ instead, but actually if two methods are "equally specialized", then there is no reason for stating that one should precede the other in the method lookup. So $=_{\mathcal{M}}$ is just a case of $\not\sim_{\mathcal{M}}$. Similar considerations apply to the cells $\{P \not\sim P, R \not\sim R\}$, $\{P \not\sim P, R = R\}$, $\{P = P, R \not\sim R\}$. For the cell $\{P < P, R \not\sim R\}$, we couldn't put the more appealing $<_{\mathcal{M}}$, otherwise $<_{\mathcal{M}}$ itself wouldn't provide us with a partial order (the refl. closure), and a topological sorting wouldn't be available for building method linearizations. Just for symmetry, we set cell $\{P > P, R \not\sim R\}$ by $\not\sim_{\mathcal{M}}$, even if $>_{\mathcal{M}}$ would be harmless in that position. The property of $<_{\mathcal{M}}$ being a *po* is important also for the alternative algorithm we show in App. A.

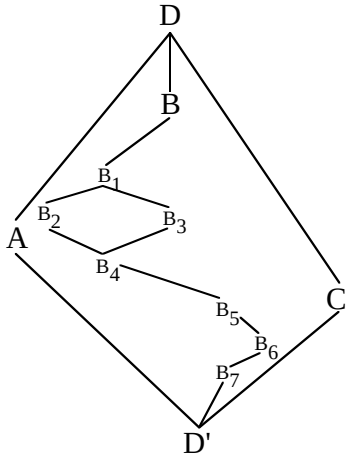
Clearly the reflexive closure of $<_{\mathcal{M}}$ is a partial order. Given a class $C$, the related class linearization is a total linear order of its superclasses. So, given a method $\mathtt{m}$ defined or directly inherited in $C$ we can obtain the $C{::}\mathtt{m}$'s method linearization by means of a *topologic sorting* of the superclasses of $C$. The sorting is performed over the "methods graph", made by the classes in the $C$-linearization, displayed by the $<_{\mathcal{M}}$ ordering. When a selection among several "equally right" nodes is needed, it is operated basing on the precedence relation in the $C$-linearization. Fig. 5 provides an example, where the class hierarchy of Fig. 3 is managed: let us assume that $\mathtt{m}_S$, represents the method $\mathtt{m}$ defined in class $S$; hence relations such

[6]They are strange occurrences in an object-oriented respect, since they are methods in a hierarchy where to a less (resp. more) specialized argument corresponds a more (resp. less) specialized result.

$\mathtt{m}_A \not\sim *$ (incomparable with all the others)
$\mathtt{m}_C \not\sim *$
$\mathtt{m}_{B_7} <_{\mathcal{M}} \mathtt{m}_{B_5}$
$\mathtt{m}_{B_2} \not\sim_{\mathcal{M}} \mathtt{m}_{B_3}$
$\mathtt{m}_{B_6} <_{\mathcal{M}} \mathtt{m}_{B_5}$
$\mathtt{m}_{B_5} <_{\mathcal{M}} \mathtt{m}_{B_4}$




(inheritance graph ex Fig. 3)




(methods graph)

Figure 5: method linearization of m from Fig. 3

as $\mathtt{m}_{B_4} <_{\mathcal{M}} \mathtt{m}_{B_2}$, or $\mathtt{m}_{D'} <_{\mathcal{M}} \mathtt{m}_{B_7}$, do trivially hold by definition of inheritance. In the figure we provide some additional relationships among methods (computed basing on the rules of Tab. 3) and show the related *methods graph*. The breadth-first based $D'$-linearization is $\{D',\ A,\ B_4,\ B_5,\ B_7,\ C,\ B_2,\ B_3,\ B_6,\ B_1,\ B,\ D\}$. The method linearization is obtained by topologically sorting the methods graph of Fig. 5: $\mathcal{M}^{D'}_{\mathtt{m}} = \{D',\ A,\ B_7,\ C,\ B_6,\ B_5,\ B_4,\ B_2,\ B_3,\ B_1,\ B,\ D\}$.

"By construction", we have the following property of method linearizations.

**Proposition 2 (property of the method linearization)**
In a method linearization $\mathcal{M}^C_{\mathtt{m}}$ a class $S_1$ precedes another $S_2$ iff
either $S_1{::}\,\mathtt{m} <_{\mathcal{M}} S_2{::}\,\mathtt{m}$
or $S_1{::}\,\mathtt{m} \not\sim_{\mathcal{M}} S_2{::}\,\mathtt{m}$ and $S_1$ precedes $S_2$ in the $C$-linearization.

△

### 3.4 Multiple dispatch based on method linearization

Let $\mathcal{M}^C_{\mathtt{m}}$ denote the $C{::}\mathtt{m}$ method linearization. In the rest of the paper by "multiple dispatch" we will mean the method retrieving executed by the following *lookup*$^m$ function.

Given a statically correct method invocation `o.m(q)`, *lookup*$^m$ returns an executable method selected from the ordered intersection $\mathcal{S}^{\underline{\mathcal{O}},\mathcal{O},\overline{\mathcal{O}}} \overset{\mathcal{L}}{\cap} \mathcal{M}^{\mathcal{O}}_{\mathtt{m}}$. This search space is a list passed to *lookup*$^m$ for the parameter $\mathcal{S}$. Note that it includes only classes with a definition for m, taken in the order of the method linearization of

the receiver's actual class. The method returned is the most specialized applicable method, in accord with the property of Lemma 1.

**Definition 5 (*lookup*$^m$, with abstraction level and method linearization)**

*lookup*$^m$ (o, m, q, $\mathcal{S}$) =
  **if** *car*($\mathcal{S}$)::m is applicable with argument q
    **then return** *car*($\mathcal{S}$)::m
    **else return** (*lookup*$^m$(o, m, q, *cdr*($\mathcal{S}$)))
**end** ***lookup***$^m$

□

**Lemma 1 (property of the multiple dispatch)**
Given a method invocation o.m(q), statically correct. Let $\overline{O}$ be the abstraction level, and $O$ the static class of the receiver. Let $\underline{O} \leq O$ be the actual class of o when the method invocation is executed. Then *lookup*$^m$, selects an applicable method $S$::m for execution, such that $\forall C \in \mathcal{S}^{\underline{O},O,\overline{O}}, C \neq S$

$$\begin{gathered}\text{either } C{::}\,\mathtt{m} \text{ is not applicable or}\\ C{::}\,\mathtt{m} \not\sim_{\mathcal{M}} S{::}\,\mathtt{m} \text{ or}\\ S{::}\,\mathtt{m} <_{\mathcal{M}} C{::}\,\mathtt{m}.\end{gathered}$$

***Proof***

The multiple dispatch checks the applicability of the $C{::}\,\mathtt{m} \in \mathcal{S}^{\underline{O},O,\overline{O}} \overset{\mathcal{L}}{\cap} \mathcal{M}^{\underline{O}}_{\mathtt{m}}$, i.e. it works in the subset of the search space made of all the classes where a definition of m appears. And it works by following the order suggested by the method linearization. So, from Prop. 2 the thesis comes. △

**Proposition 3 (run-time correctness of a method invocation)**
Given a method invocation o.m(q), statically correct. Let $\overline{O}$ be the abstraction level, and $O$ the static class of the receiver. Let $Q$ be the static class of q and $R$ the result static class. Let $\underline{O} \leq O$ be the actual class of o, and $\underline{Q}$ the actual class of q when the method invocation is executed.

Then there is at least one method executable at run-time and, whatever is the method actually selected for execution,

1. it returns an object of class $R' \leq R$;
2. there is no more specialized method applicable.

***Proof***

Since the invocation is statically correct, the method $\overline{O}$::m(p:$P_{\overline{O}}$):$R_{\overline{O}}$ is statically applicable. So, in the hypothesis it is $R = R_{\overline{O}}$.

The execution of $\mathtt{o}^{\underline{O}}\mathtt{.m(q}^{\underline{Q}}\mathtt{)}$, is performed by multiple dispatch on the method linearization $\mathcal{M}_{\mathtt{m}}^{\underline{O}} = \{\ \underline{O},\ O_1,\ \ldots,\ O_k,\ \overline{O},\ O_{k+1},\ \ldots,\ O_n\ \}$ (without any loss of generality, we have assumed that there is an `m` definition in $\underline{O}$).

*lookup*$^m$ ranges over the method linearization, where *lookup*$^{dd}$ did on the plain linearization: define $\mathcal{S} = \mathcal{S}^{\overline{O},O,\underline{O}} \overset{\mathcal{L}}{\cap} \mathcal{M}_{\mathtt{m}}^{\underline{O}}$. Then, $\mathcal{S} \neq \emptyset$, and there is at least one executable method, since at least $\overline{O} \in \mathcal{S}$ and $\overline{O}\mathtt{::m}$ is applicable as it was statically applicable. Moreover,

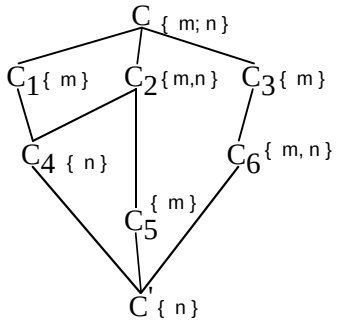
1. $\forall C \in \mathcal{S}$, it is $\underline{O} \leq C \leq \overline{O}$. So $\forall C \in \mathcal{S}$, the result of $C\mathtt{::m}$ is of class $R_C \leq R$, by the covariant rule for method redefinition.

2. That there is no more specialized method is ensured by Lemma 1.

△

### 3.5 Static method linearization

The concept of method linearization can help us in expressing the static type checking rules of Def. 3.




$$\mathcal{M}_{\mathtt{n}}^{C'} = \{C', C_4, C_6, C_2, C\}$$

$$\mathcal{SM}_{\mathtt{n}}^{C} = \{C'\}$$

$$\mathcal{M}_{\mathtt{m}}^{C'} = \{C_6, C_5, C_2, C_1, C_3, C\}$$

$$\mathcal{SM}_{\mathtt{m}}^{C'} = \{C_6, C_5, C_1\}$$

Figure 6: an example of static method linearization

So far we have been talking about the method lookup that acts at run-time, when code must be selected for execution.

At compile-time, in order to type check a method invocation `o.m(q)`, a sort of method lookup is still performed, but a method is selected to state the abstraction level of the expression. Applicability here is based on the static classes of receiver and arguments. This method lookup can be performed by *lookup*$^m$, once the list of classes where to lookup is provided by the *static method linearization* $\mathcal{SM}_{\mathtt{m}}^{C}$, actually a subset of $\mathcal{M}_{\mathtt{m}}^{C}$, built from Def. 3. See Fig. 6 for an example.

The class $C'$ defines explicitly only the method `n`, while its superclasses define several `n` and `m`. Now suppose that `o.n(p)` is the method invocation to type check

and that the method linearization provided by $C'$ is $\mathcal{M}_{\mathtt{n}}^{C'} = \{C', C_4, C_6, C_2, C\}$ (only the classes in which an effective definition of `n` occurs). By applying the rules of Def. 3 the only class to check for an applicable method is $C'$, since there is a method `n` defined there. So $\mathcal{SM}_{\mathtt{n}}^{C'} = \{C'\}$. On the other hand, if the method invocation to type check were `o.m(q)`, suppose is $\mathcal{M}_{\mathtt{m}}^{C'} = \{C_6, C_5, C_2, C_1, C_3, C\}$; then the static method linearization is $\mathcal{SM}_{\mathtt{m}}^{C'} = \{C_6, C_5, C_1\}$ (note that $C_2$ is covered by $C_5$). And, the looking for an applicable method only in the classes of $\mathcal{SM}_{\mathtt{m}}^{C'}$ implements the rules of Def. 3.

In the following we provide a restatement of the rules for the static correctness of a method invocation, such that the static method linearization is used by a suitable lookup algorithm.

**Definition 6 (static method linearization)** Let $\mathcal{M}_{\mathtt{m}}^{O}$ be the method linearization associated to `m` in $O$. The *static method linearization* $\mathcal{SM}_{\mathtt{m}}^{O}$ is the subset of $\mathcal{M}_{\mathtt{m}}^{O}$ composed by all classes such that no subclass (until $O$, comprised) contains an explicit definition of `m`. It can be obtained, from $\mathcal{M}_{\mathtt{m}}^{O}$, by the following function:

*static-method-linearization* $(\mathcal{M}_{\mathtt{m}}^{O}) =$
  $\mathcal{M} := \mathcal{M}_{\mathtt{m}}^{O}$;
  $\mathcal{R} :=$ **nil**;
  $\forall C \in \mathcal{M}$
    $\forall D \in \mathcal{M}$
      **if** $D \in C$-{inh-list} **then** mark $D$ *covered*;
  $\forall C \in \mathcal{M}$
    **if** $C$ is *covered* **then** *enqueue*($C$, $\mathcal{R}$);
  **return**($\mathcal{R}$);
**end** *static-method-linearization*

□

**Definition 7 (static method lookup)** Let `o.m(q)` be a method invocation, with `o`:$O$ and `q`:$Q$. Let $\mathcal{M}_{\mathtt{m}}^{O}$ be the method linearization associated to `m` in $O$. Let the $\mathcal{SM}_{\mathtt{m}}^{O}$ = *static-method-linearization* $(\mathcal{M}_{\mathtt{m}}^{O})$. The following function returns the abstraction level of the method invocation (the class in $\mathcal{SM}_{\mathtt{m}}^{O}$ that makes the invocation statically correct, or **none** if the invocation is not statically correct):

*static-method-lookup* (`m`, $O$, $Q$, $\mathcal{SM}_{\mathtt{m}}^{O}$) =
  **if not** *empty-list*($\mathcal{SM}_{\mathtt{m}}^{O}$)
    **then**
      **if** *car*($\mathcal{SM}_{\mathtt{m}}^{O}$)`::m` is applicable with argument of class $Q$
        **then return**(*car*($\mathcal{SM}_{\mathtt{m}}^{O}$));
        **else return**(*static-method-lookup* (`m`, $O$, $Q$, cdr($\mathcal{SM}_{\mathtt{m}}^{O}$)));
    **else return**(**none**) ;
**end** *static-method-lookup*

□

**Definition 8 (static correctness through static method linearization)**
Let `o.m(q)` be a method invocation to be statically type checked, with `o`:$O$ and `q`:$Q$. Let $\mathcal{M}^O_{\mathtt{m}}$ be the method linearization associated to `m` in $O$. Let $\mathcal{SM}^O_{\mathtt{m}} =$ *static-method-linearization* ($\mathcal{M}^O_{\mathtt{m}}$). Let $S =$ *static-method-lookup* (`m`, $O$, $Q$, $\mathcal{SM}^O_{\mathtt{m}}$): `o.m(q)` is statically correct, with abstraction level equal to $S$, iff $S \neq$ **none**.

□

### 3.6 Pit-stop

In this section we have considered the use of covariant method redefinition in the framework of the specialization inheritance defined in Sec. 1.1. In this respect, the features of our proposal are as follows:

- it defines a multiple dispatch mechanism that is sensitive to the abstraction level (i.e. to the static characteristics) of the method invocation;
- it eliminates the previously needed constraints and enables the greater flexibility of the specialization inheritance mechanism; (in particular, it guarantees the coherence of the static and run-time result classes);
- it ensures that the method selected for execution is such that never another more specialized method could have been selected, among the applicable ones.

For a comparison of ours with other approaches to multiple dispatch, see App. B.2. Here we only stress that, w.r.t. other authors solutions and to proposals made in collaboration by the author, we relax the constraints on method redefinition, bound more carefully the method lookup search space by the double diamond, and apply the method linearization technique.

## 4 Instance variables redefinition

Here we confront the problem arisen in Fig. 2, case (b).

The *Student*`::rank` instance variable has been redefined in *Special_Student*, narrowing its type from *Evaluation* to *Ext_Evaluation*. Also the `Set_rank` method has been consequently redefined, but this further specialization of the updating method does not protect from type errors at run-time, once polymorphic assignments have been performed. In Fig. 2, case (b), when `t.Interrogate?(s)` is executed, `t` refers to an object of class *Teacher*, and `s` polymorphically refers to a special student. Then, by multiple dispatch, *Teacher*`::Interrogate?` is selected to execute. It assigns an *Evaluation* object to the *Ext_Evaluation* instance variable of the special student, dishing out a type failure.

This problem doesn't depend on the existence of covariantly redefined methods *per sé*. It depends on the presence of covariantly redefined instance variables, together with particular methods that update such variables. We call this problem *update problem*.

(In a class, we call *update method* any method which performs an explicit update, such as `x:= ···`, over an instance variable of the class[7]).

**Definition 9 (update problem)** Given class $A$ and $B$,

| | |
|---|---|
| class $A$ {} | class $B$ { $A$} |
| `x:`$X$`;` | `x:`$X'$ |
| `m(p:`$X$`):`$R$ is {··· `x:=p` ···}; | endclass |
| endclass | |

with $B$ redefining an instance variable `x` of $A$, and $A$ defining a method whose instructions update `x`, then, the execution of the polymorphic method invocation $\mathtt{a}^B\mathtt{.m(q}^X\mathtt{)}$ incurs in a run-time type error (the update problem).

□

In the definition we clearly suppose that `a` and `q` are expressions of static class resp. $A$ and $X$. The type failure in $\mathtt{a}^B\mathtt{.m(q}^X\mathtt{)}$ is taken while executing the assignment `x:=p` in `m`'s definition (i.e. while doing `a.x:=q` with `a` of class $B$ and `q` of class $X$): $\mathtt{x}^{X'}\mathtt{:=}\ \mathtt{q}^X$ is an illegal run-time update that appeared safe at compile-time. (Whether a redefinition of `m` is given or not in $B$ isn't significant: if it were naturally specialized it wouldn't be applicable).

This problem occurs in any language where instance variable redefinition is allowed independently of the adopted polymorphism discipline (subtyping or inheritance bounded). It occurs in Loglan ([15]), Eiffel ([16]) and in any language allowing a "left-value" be assigned by a subclass object. It occurs also in CLOS, so as in other languages following a multiple dispatch mechanism. In general, if we allow covariant redefinition of instance variables, it can't be ensured that a "super-method" (a method defined in a superclass) is type safe when executed for a method invocation sent to an object of subclass.

Then, an unconstrained "specialization bounded polymorphism" can be type unsafe. Yet we see that there are many cases in which polymorphism is possible in our permissive inheritance and we would like to let them be supported. Some plain cases are discussed in [11, 10]: they are simple, there is no update problem, so the use of our multiple dispatch is enough to allow their programming. It comes out that we can give some conditions to let also more complicate cases, like our "polymorphic school", be supported.

As we did for method redefinition in Sec. 3, here we try to see the edge where the major expressive power of the specialization inheritance still allows for statically checkable type safeness. Note that a very first solution would be to forbid any polymorphic instructions when the conditions of Def. 9. This would be quite strong. The developers of Eiffel are working on a lighter solution: they forbid the

[7] Usually such methods are of the `Set_x` kind seen in previous example, i.e. is a method "dedicated" to the update.

"catcalls" (i.e. the above troublesome method invocations) only when an analysis of the program shows that the receiver object has been assigned polymorphically ([17, 18]).

In the following we show that some more cases are safely recoverable at cost of asking the programmer for some additional work: the prevention against update problems is the use of suitable *default-value methods*.

### 4.1 Default-value methods

Still referring to our error prone method invocation in Fig. 2, case (b), the starting observation is that the abstraction level of `t.Interrogate?(p)` is *Teacher*, and in *Teacher*`::Interrogate?` the abstraction level of `p.Set_rank(e)` is *Student*. Everything is statically correct here. At run-time we have that type safeness follows, provided that:

1. After the redefinition of `rank` in *Special_Student*, there is a method *Special_Student*`::default-value(x:`*Evaluation*`):`*Ext_Evaluation* that returns a value of class *Ext_Evaluation* (the class of *Special_Student*`::rank`), computed from a value of class *Evaluation* (the class of *Student*`::rank`).

2. Each time *Special_Student*`::rank` is going to be assigned by a plain *Evaluation* value, the assignment operator is able to invoke that default-value method, and assign *Special_Student*`::rank` by the returned *Ext_Evaluation* value.

---

```
class Student { Person}
...
rank:Evaluation                                  // Evaluation = {mark:integer}
Set_rank(p:Evaluation) is {assign(rank, p)}
endclass

class Special_Student {Student}                  // Ext_Evaluation =
rank:Ext_Evaluation                              // {mark:integer; ability:string}
default-value(x:Evaluation):Ext_Evaluation
  is {result := new Ext_Evaluation;
    result.mark := x.mark;
    result.ability := "no comment";}
Set_rank(p:Ext_Evaluation)
  is {assign(rank, p); };
endclass
```

Figure 7: new students for the polymorphic school

---

This is just a sketch of the needed rules and needs further development, but now we can define alternatively the student classes we used so far: in Fig. 7

*Special_Student* defines an additional default-value method, able to compute an *Ext_Evaluation* value from an *Evaluation* one. The computed value is "standard" but it is related to the given *Evaluation* value. So, we can say that the default-value method provides a special student with the most suitable (specialized) standard value that s/he needs when acting at a *Student* abstraction level. The other difference in Fig. 7 is that the methods use a different assignment operator. `assign` is foreseen as a method, common to all classes, that is devised to execute safe assignments, possibly by calling the suitable default-value method when needed. Here we provide the reader with a preliminary definition, to proceed with an example of its use. Some peculiarity, not important at now, will be fixed in the final definition.

**Definition 10 (`assign` method - provisional definition)** Let $\leq$ be the subclassing relation, and `:=` the usual assignment operator. We assume that there is a "system-class" *Top* automatically inherited by any other class. The only contents of *Top* is the method `assign`, which hence is an inherited attribute for any class.

```
assign(x, y:Top) =
    if classof(y) ≤ classof(x)
       then x := y
       else self.assign(x, self.default-value(y))
end
```

□

**Example 1 (Execution of `t.Interrogate?(s)`)** Here we trace the execution of the method invocation $\texttt{t}^{Teacher}\texttt{.Interrogate?(s}^{Special\_Student}\texttt{)}$ we met above.

- By multiple dispatch, the method *T*eacher`::Interrogate?` is selected.
- Then $\texttt{p}^{Special\_Student}\texttt{.Set\_rank(e}^{Evaluation}\texttt{)}$ is executed, leading to the execution of
- $\texttt{assign(p.rank}^{Evaluation}\texttt{, e}^{Ext\_Evaluation}\texttt{)}$. The assignment `p.rank := e` clearly can't stand so the **else** part of `assign` is executed.
- For `assign(p.rank, self.default-value(e))`, first `self.default-value(e)` is executed. *Special_Student*`::default-value` is found applicable and executed; it returns an object $\texttt{o}^{Ext\_Evaluation}$.
- Then the outer $\texttt{assign(p.rank, o}^{Ext\_Evaluation}\texttt{)}$ is executed happily (**then** branch).

♢

So far we have given the intuition of our treatment of polymorphic instructions in presence of instance variable redefinition. The following definitions state the requirements for managing safe polymorphism in a specialization inheritance environment, where instance variable redefinitions occur. Def. 11 defines constraints rules to be followed during the programming. Def. 12 extends and replaces Def. 10,

to express the accomplishments that the programmer can expect provided by the type system. Immediately after the definitions, some note follows to explain their rationale.

**Definition 11 (constraints for instance variable redefinition)** Given $C$,

$$\begin{array}{l}\underline{\text{class}}\ C\ \{C_1, \ldots, C_n\}\\ \mathtt{v}_1\mathtt{:}V_1, \cdots, \mathtt{v}_n\mathtt{:}V_n\\ \mathtt{m}_1\mathtt{(p:}P_1\mathtt{)}\mathtt{:}R_1\ \underline{\text{is}}\ \{\cdots\}; \cdots \mathtt{m}_l\mathtt{(p:}P_l\mathtt{)}\mathtt{:}R_l\ \underline{\text{is}}\ \{\cdots\};\\ \underline{\text{endclass}}\end{array}$$

1. for each instance variable $\mathtt{v}_i\mathtt{:}V_i$ that redefines a directly inherited $\mathtt{v}_i\mathtt{:}V_i^{super}$ there must be in $C$ a method $\mathtt{v}_i\mathtt{-default\text{-}value(p:}V_i^{super}\mathtt{)}\mathtt{:}V_i$.
2. Any update method in $C$ can update instance variables exclusively by the `assign` method.

□

**Definition 12 (`assign` and `default-value` methods)**
Let $\leq$ be the subclassing relation, and `:=` the usual assignment operator.

We assume that there is a "system-class" *Top*, as the topmost class of any class hierarchy. The only contents of *Top* is the following method `assign`, which hence is an inherited attribute for any class. In `assign`, `x` is the *left-value* being assigned by `y` and `Id` is the string containing `x`'s identifier.

```
assign(x, y:Top;  Id:string) =
   if classof(y) ≤ classof(x)
      then x := y
      else
        name := concatenate(Id, "-default-value")
        self.assign(x, self.name(y), Id)
end
```

Any invocation of the method `assign` has abstraction level equal to *Top*.

For the default-value method invocation in assign (i.e. for any default-value method invocation) the abstraction level is not computed and stated by definition as equal to *Top*. □

Remember that `assign` must be used exclusively for updating instance variables: so the admitted `x` are of the form either `IVar` or `Id.IVar`, where `IVar` stands for an instance variable identifier, and `Id` for a variable identifier[8].

W.r.t. Def. 12, two aspects must be stressed, both regarding the default-value method invocation: its form and its abstraction level.

[8] If the good principle of updating instance variables only by `Set` methods is followed, then `x` is always of the `IVar` kind.

1. In a general case of class definition, there might be several instance variables of the same class, redefining previous ones of the same class: say `a:`$X'$, `b:`$X'$, ..., `z:`$X'$ redefine previous `a:`$X$, `b:`$X$, ..., `z:`$X$. If there is only one `default-value(p:`$X$`):`$X'$ available, we have to use it for all such instance variables, while we might want to be able to express different ways of recovering a default value for the different variables. We can get much more expressive power in our class definitions, if we can support such instance-variable-custom default-value methods. This is pursued through the (admittedly awkward) extension of the default-value method name, by the variable name, in Def. 11 and the according changes in the definition of `assign` (Def. 12).

2. What is more awkward is that for evident reasons we cannot compute an abstraction level for the `self.name(y)` occurring in the last instruction of `assign`, since there isn't a method name at compile-time in that method invocation. We stand with it, by stating that *Top* is the abstraction level. In this way we don't care about the class of the resulting object. But we are ensured, by the constraints of Def. 11, 1) that such a default-value method will be retrieved at run-time, and 2) that the instance variable will be assignable by the resulting object.

   The default-value methods on their own, when requested by Def. 11 (and *so* defined) are type checked (and executed) as normal methods.

**Example 2** Suppose that `a`$^C$`.Set_x(q`$^{X_A}$`)` is to be executed, in the context provided by Fig. 8.

---

class $A$ { }
`x:`$X_A$
`Set_x(p:`$X_A$`)` is {`self.assign(x, p, "x")`}
endclass

class $B$ { $A$}
`x:`$X_B$
`x-default-value(p:`$X_A$`):`$X_B$ is {$\cdots$}
endclass

class $C$ { $B$}
`x:`$X_C$
`x-default-value(p:`$X_B$`):`$X_C$ is {$\cdots$}
endclass

`a:`$A$`;`
`q:`$X_A$`;`
`a := new` $C$`;`
`q := new` $X_A$`;`

Figure 8: use of assign

---

Tab. 4 points out the static and dynamic characteristics of the invocation. The column labeled AL points out the abstraction level of the involved method invocations, the one labeled R/T points out the run-time class of the method invocation receiver and the FROM column shows for each invocation the class from where the method to be executed is selected.

The flow of method invocations is shown in the call order (from the top). Note that $\dagger'$ and $\ddagger'$ are just restatements of older invocations (resp. $\dagger$ and $\ddagger$), after that some arguments have been computed. That's why we put them between parentheses and don't repeat the AL, R/T and FROM data.

| | method invocation | AL | R/T | FROM |
|---|---|---|---|---|
| ¶ | `a.Set_x(q)` | *A* | *C* | *A* |
| $\star$ | `self.assign(a.x,q,"x")` | *Top* | *C* | *Top* |
| $\dagger$ | `self.assign(a.x,self.x-default-value(q),"x")` | *Top* | *C* | *Top* |
| $\flat$ | `self.x-default-value(q)` | *Top* | *C* | *B* |
| $\dagger'$ | ( `self.assign(a.x,`$o^{X_B}$`,"x")` ) | | | |
| $\ddagger$ | `self.assign(a.x,self.x-default-value(`$o^{X_B}$`),"x")` | *Top* | *C* | *Top* |
| $\sharp$ | `self.x-default-value(`$o^{X_B}$`)` | *Top* | *C* | *C* |
| $\ddagger'$ | ( `self.assign(a.x,`$o^{X_C}$`,"x")` ) | | | |

Table 4: `a.Set_x(q)` in Fig. 8.

An intuitive description of the invocation follows:

- After ¶, `assign(a.x, q, "x")` is executed in $\star$, but the assignment can't take place, because `a.x` is a variable of class $X_C$ and `q` refers to an object of class $X_A$. So

- `assign(a.x, self.x-default-value(q), "x")` ($\dagger$) is executed.

- The assigning object in $\dagger$ is to be computed by $\flat$: the method lookup checks that $C$`::x-default-value` is not applicable, while $B$`::x-default-value` is applicable; it is selected and returns an object `o` of class $X_B$ ($o^{X_B}$). Then $\dagger$ is restated in $\dagger'$.

- `assign(a.x,` $o^{X_B}$`)` ($\dagger'$) is executed. But the assignment can't take place; so

- `assign(a.x, self.x-default-value(`$o^{X_B}$`), "x")` is executed ($\ddagger$). The assigning object is computed by $\sharp$: $C$`::x-default-value` is applicable; it is executed and returns an object of class $X_C$ ($o^{X_C}$).

  Then the assignment $\ddagger$ is restated in $\ddagger'$.

- `assign(a.x,` $o^{X_C}$`, "x")` ($\ddagger'$) assigns `a.x` by $o^{X_C}$.

$\diamondsuit$

To see the correctness of our approach we have to show that when instance variables are redefined accomplishing the requirements of Def. 11 in programs with update problems, troublesome updates of instance variables are always positively resolved (i.e. suitable default-value methods are found, they are applicable and let the assignment be executed without type failures).

**Proposition 4 (correctness of update statements)**

Let $\texttt{o}^C\texttt{.m(q}^X\texttt{)}$ be an update method invocation, statically correct, with abstraction level $A$.

Let `assign(x,q,"x")` be the update being performed, with `x` an instance variable with (re)definition $\texttt{x:}X_A$ in a class $A$ and redefinition $\texttt{x:}X'$ either in $C$ or in a class $S$, $C < S < A$.

Let the constraints on the instance variable redefinition (Def. 11) be fulfilled.

Then the update is executed without type errors, after the execution of at least the `x-default-value` method related to the redefinition $\texttt{x:}X'$.

***Proof***

See Prop. 5. So far we have been dealing only with single inheritance, while there the same property is proven for the general case of multiple inheritance hierarchies. △

## 4.2 Default-value methods and multiple inheritance

The application of the default-value methods technique produces some strange effects in case of multiple inheritance. In Fig. 9 we see that if a subclass redefines an instance variable inherited by several superclasses, several default-value methods must be defined and we have that the subclass features a multiple method.

(By *multiple method* we mean the several definitions for the same method occurring in the same class.)

---

$A$, $B$ and $C$ are the classes of Fig.8;

class $D$ { $B$, $C$ }
$\texttt{x:}X_D$
$\texttt{x-default-value(p:}X_B\texttt{):}X_D$ is $\{\cdots\}$
$\texttt{x-default-value(p:}X_C\texttt{):}X_D$ is $\{\cdots\}$
endclass

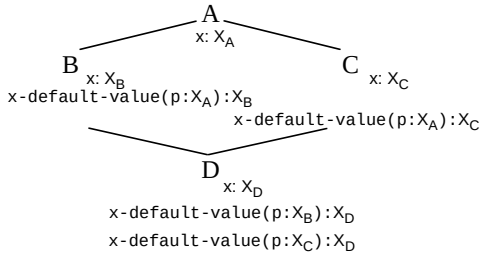



Figure 9: several default-value methods induced by multiple inheritance

---

For dealing with the multiply defined default-value methods of Fig. 9 we have to establish the *intended semantics* of such a multiple definition embedded in a class.

**Definition 13 (intended semantics of multiple default-value methods)**

Let $C$ be a class with multiple definitions for `x-default-value` (say $\texttt{x-default-value}^1$, ..., $\texttt{x-default-value}^n$).

Whenever a method invocation `self.x-default-value(q)` is going to be executed and $C$ is in the search space for $lookup^m$ ($C \in \mathcal{S}^{\underline{O},O,\overline{O}}$), then $\forall i = 1 \cdots n$ the definition `x-default-value`$^i$ must be available for checking its applicability and for possibly selecting it.

Moreover, the sequence of the given `x-default-value`$^i$ definitions, provides the basic linearization ordering to compute $\mathcal{M}^C_{\texttt{x-default-value}}$.

□

So we intend that the definitions in $C$ must be all available to instances of $C$ or subclasses, in order to compute instance variable values: whenever one of them is in the scope of $lookup^m$, its companions would be in the scope as well. Moreover, since $lookup^m$ works with method linearizations, and to compute a method linearization needs a basic ordering for unrelated methods, we intend that the sequence of the `x-default-value`$^i$ definitions is significant.

We can implement such intended semantics without real modifications to the class construct and to the multiple dispatch mechanism as presented so far. What we do is to perform a class transformation that creates new classes for the multiply defined methods and let them be directly inherited by $C$: the effect of multiply defined default-value methods, is gained by multiple inheritance. It is the transformed hierarchy that is used for static type checking and multiple dispatch purposes.

**Definition 14 (Class hierarchy transformation)**
Given $C$, a class with inheritance list $C::$*inh-list*, with redefinition `x:`$X_C$, and definitions `x-default-value`$^1$`(p:`$X^1$`):`$X_C$, …, `x-default-value`$^n$`(p:`$X^n$`):`$X_C$, then the class hierarchy owning $C$ is transformed as follows:

**(interface)** A new class $\overline{C}$ is created, such that $\overline{C}::$*inh-list* $= C::$*inh-list*, and

1. all the attributes defined in $C$ and used by an `x-default-value`$^i$ (with $i \in [1 \cdots n]$) are placed in $\overline{C}$;
2. the redefinition `x:`$X_C$ is placed in $\overline{C}$.

**(super)** $n$ new classes are created, $C_1$, …, $C_n$, such that, for $i = 1 \cdots n$,

1. $C_i::$*inh-list* $= \{\overline{C}\}$;
2. the only attribute of $C_i$ is the `x-default-value`$^i$ definition.

**($C$ modif.)** $C$ is modified such that $C::$*inh-list* $= \{C^1, \ldots, C^n\}$, and

1. its definitions `x-default-value`$^i$ ($i = 1 \cdots n$) are removed;
2. its attributes that are used by an `x-default-value`$^i$ (with $i \in [1 \cdots n]$) are removed;

□

By such transformation, from a starting hierarchy as in Fig. 9, we obtain the effective hierarchy of Fig. 10.

W.r.t. Fig. 9, this is the class hierarchy effectively used for type checking and method lookup purposes. The classes $\overline{D}$, $D_1$, $D_2$ are added; they are never used by the programmer to declare variables or to instantiate objects.

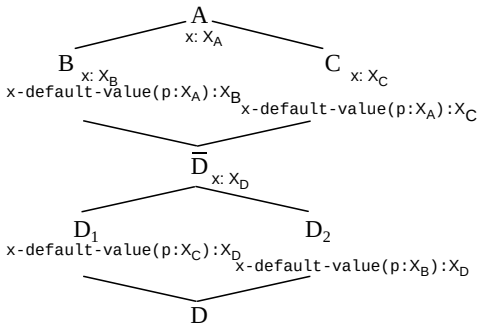



Figure 10: the effective hierarchy obtained by Fig. 9

In the figure, the classes $D_1$, $D_2$ and $\overline{D}$ are added and $D$ is supposed to slightly change, as it follows:

| | |
|---|---|
| class $\overline{D}$ { $B$, $C$ } | class $D_1$ { $\overline{D}$ } |
| endclass | `x-default-value(p:`$X_B$`):`$X_D$ is $\{\cdots\}$ |
| | endclass |
| class $D$ { $D_1$, $D_2$ } | class $D_2$ { $\overline{D}$ } |
| `x:`$X_D$ | `x-default-value(p:`$X_C$`):`$X_D$ is $\{\cdots\}$ |
| endclass | endclass |

Notice that the changes in the inheritance list of $D$ turn out to be uneffective on its semantics: the shape of the instances of $D$ is the same: the `x-default-value` methods are now inherited from direct superclasses and will be all available at runtime, since the abstraction level of their invocation is always *Top*: so the modification of their position in the hierarchy is ineffective on the behavior they implement.

We could assume that, once the programmer has defined multiple default-value methods, in a given sequence in $D$, this same ordering is used for setting up the additional immediate predecessors of $D$. Basing on this primitive ordering, a method linearization, can be computed also for multiple default-value methods. Looking at Fig. 9 we can expect well that $\mathcal{M}^D_{\texttt{x-default-value}}$ does enlist (suppose $X_C \leq X_B$) the first and second *x-default-value* defined in $D$, then the ones defined in $C$ and $B$, in this order. And, from Fig. 10, it is $\mathcal{M}^D_{\texttt{x-default-value}} = \{D_2, D_1, C, B\}$.

To see the correctness of our approach we have to show that after the class transformation our multiple dispatch fulfills the intended semantics of Def. 13, and that in programs with update problems, troublesome updates of instance variables are always positively resolved.

**Proposition 5 (correctness of update statements)**
Let $\texttt{o}^C\texttt{.m(q}^X\texttt{)}$ be an update method invocation, statically correct, with abstraction level $A$.

Let `assign(x,q,"x")` be the update being performed, with `x` an instance variable of class $X$ in $A$ and of class $X'$ in $C$.

Let the constraints on the instance variable redefinition (Def. 11) be fulfilled.

Then the update is executed without type errors, after the execution of at least the `x-default-value` method related to the redefinition `x:`$X'$.

***Proof***

We assume to work on a multiple inheritance hierarchy. So a redefinition of `x` may involve several `x-default-value`$^i$ definitions.

That `x` is of class $X'$ in $C$, means that there is a class $S$, with $C \leq S < A$ where a redefinition `x:X'` took place, such that no other redefinitions occur between $S$ and $C$. Since Def. 11 is fullfilled $S$ (or $\overline{S}$ after transformation) directly provides $C$ with the needed `x-default-value`$^i$ methods. So we can assume that the redefinition `x:`$X'$ takes place in $C$ ($C = S$) without loos of generality.

Assume there are definitions

`x-default-value(p:`$X^1$`):`$X'$, ..., `x-default-value(p:`$X^n$`):`$X'$, in $C$ ($\overline{C}$).

A characteristics of $\mathcal{M}^C_{\texttt{x-default-value}}$ is that it is equal to the sequence of the $C^i$s (ordered suitably) followed by $\mathcal{M}^{\overline{C}}_{\texttt{x-default-value}}$. This is true because the return type of any `x-default-value`$^i$ is $X'$, and all the `x-default-value` methods defined over $\overline{C}$ return the class more general definitions of `x`.

Under the hypotheses `o.x` is an instance variable of class $X'$, being assigned by an object of class $X \geq X'$. Then `assign` calls for `x-default-value` dispatching. The lookup is in $\mathcal{M}^C_{\texttt{x-default-value}}$ and starts looking in the $C^i$.

1. If between $A$ and $C$ there are no other redefinitions for `x`, we can assume that there is a $k$ such that $X^k = X'$, so there is the $C^k$`::x-default-value(p:`$X$`):`$X'$ available for execution. It returns an object of class $X'$ that can be assigned to `o.x`.

2. If between $A$`::x:`$X$ and $C$`::x:`$X'$ there are redefinitions for `x`, then we can show that `assign(x`$^{X'}$`,q`$^X$`,"x")` at last produces an invocation `assign(x`$^{X'}$`,q`$^Y$`,"x")` such that there exist a $C^i$`::x-default-value(p:`$Y$`):`$X'$. (So the real assignment can take place after one more default-value method invocation.)

   (a) Assume there is one only redefinition `x:`$X_{S_1}$ in class $S_1$ ($\overline{C} < S_1 < A$). Then, there are a $C^k$`::x-default-value(p:`$X_{S_1}$`):`$X'$ and a $S_1^h$`::x-default-value(p:`$X$`):`$X_{S_1}$ (by Def. 11).
   Since in `assign(x`$^{X'}$`,q`$^X$`,"x") x:=q` can't take place, the assignment is tried by `::x-default-value(q)`.
   If a $C^i$`::x-default-value` is applicable then it produces an object of class $X'$ and `x` is succesfully assigned.
   Otherwise, surely $S_1^h$`::x-default-value` is applicable. It returns an object of class $X_{S_1}$ by which `assign` can succesfully assign `x` (by point

1.). Indeed, this time $C^k$`::x-default-value` is applicable and returns a object of class $X'$ as requested for the assignment.

So, if there is one redefinition `x:`$X_{S_1}$ between $A$ and $C$, the thesis holds.

(b) Assume that when between $A$`::x:`$X$ and $C$`::x:`$X'$ there are $m$ redefinitions in classes $S_i$, with $\overline{C} < S_m < \cdots < S_2 < S_1 < A$ then the thesis holds.

Then, assume that between $A$`::x:`$X$ and $C$`::x:`$X'$ there are $m+1$ redefinitions in classes $S_i$, with $\overline{C} < S_{m+1} < \cdots < S_2 < S_1 < A$, and that `assign(x`$^{X'}$`,q`$^{X}$`,"x")` is in execution.

`x-default-value(q)` is invocated.

If one of the $C^i$`::x-default-value` is applicable, we are done (it produces an object of class $X'$).

Otherwise, by multiple dispatch a default-value method applicable to argument of class $X$ is selected in $\mathcal{M}^C_{\texttt{x-default-value}}$ (and at least one there must be since $S_1$ redefines $A$`::x:`$X$ in `x:`$X_{S_1}$). Assume `o`$^{\overline{X}}$ is the object produced by that selected method. Note that $\overline{X}$ is the class of some redefinition of `x` after $A$ (since it has been produced by a `x-default-value` method occurring after $A$). Say it is $\overline{X} = X_{S_h}$.

So we have that the first `assign` produced an invocation `assign(x`$^{X'}$`,o`$^{X_{S_h}}$`,"x")`. This invocation is successfull by the induction hypothesis, where simply $S_h$ plays the rule of $A$.

△

We conclude this section with an example, related to Fig. 10.

**Example 3** Suppose `a`$^{D}$`.Set_x(q`$^{X_A}$`)` is to be executed, where `a` and `q` are expressions of static class, resp., $A$ and $X_A$. Then an update problem occurs and we would get type failure if suitable `x-default-values` methods were not available.

Without any loss of generality we can suppose also that $X_C \leq X_B$, so to fix a method linearization of the default-value methods in $D$: $\mathcal{M}^D_{\texttt{x-default-value}} = \{D_2, D_1, C, B\}$.

Tab. 5 shows the analysis of the execution of the method invocation, in the fashion used in Ex. 2. Further comments follow.

- The execution of ¶ implies that $\star$ is called[9].

- The assignment `a.x := q` cannot stand, since the `a.x` is actually an instance variable of class $X_D$ while `q` refers to an object of class $X_A$.

- So † is called to perform the assignment of `a.x` by `self.x-default-value(q)`.

[9]There is only one `Set_x` in this example and it is in class $A$; of course, its presence in other classes here would be harmless, if not useless

| | method invocation | AL | R/T | FROM |
|---|---|---|---|---|
| ¶ | `a.Set_x(q)` | $A$ | $D$ | $A$ |
| ⋆ | `self.assign(a.x,q,"x")` | *Top* | $D$ | *Top* |
| † | `self.assign(a.x,self.x-default-value(q),"x")` | *Top* | $D$ | *Top* |
| ♭ | `self.x-default-value(q)` | *Top* | $D$ | $C$ |
| †′ | ( `self.assign(a.x,` $\mathtt{o}^{X_C}$`,"x")` ) | | | |
| ‡ | `self.assign(a.x,self.x-default-value(`$\mathtt{o}^{X_C}$`),"x")` | *Top* | $D$ | $D$ |
| ♯ | `self.x-default-value(`$\mathtt{o}^{X_C}$`)` | *Top* | $D$ | $D_2$ |
| ‡′ | ( `self.assign(a.x,` $\mathtt{o}^{X_D}$`,"x")` ) | | | |

Table 5: Evaluation of the update method invocation in Ex. 3

- In ♭ the second argument of † is computed by the execution of the method invocation `self.x-default-value(q)`. Here the $C$`::x-default-value` is selected for application, since it is the first applicable method met along $\mathcal{M}^{D}_{\mathtt{x-default-value}}$ (neither $D_2$`::x-default-value` nor $D_1$`::x-default-value` are applicable). Its execution returns an object $\mathtt{o}^{X_C}$ (of class $X_C$).

- Then we come back to perform the assignment †′ (actually this is still †, just after having computed the needed second argument: accordingly, the AL, R/T and FROM data are not repeated).

  The assignment `a.x :=` $\mathtt{o}^{X_C}$ can't be executed as it is, so the ‡ method invocation is performed. Now, in ♯, the situation is quite similar to ♭.

- In ♯, in order to compute the second argument of ‡, the $D_2$`::x-default-value` is selected for application. Its execution returns an object $\mathtt{o}^{X_D}$ (of class $X_D$).

- Then ‡′ just continues the execution of ‡. This time the assignment we have been struggling for can take place, as `a.x :=` $\mathtt{o}^{X_D}$.

♢

### 4.3 Pit-stop

In this section we have seen that

1. actually there aren't troubles at all with covariant instance variable redefinition, while the update problem doesn't occur.

   So, for instance, if the redefined instance variable is initialized at object-creation-time and never more updated (i.e. if it is *non mutable*), then there aren't troubles.

2. If there are covariant redefinitions of instance variables and related updating methods, we can still support safe polymorphism, once the programmer has provided the suitable default-value methods.

3. As far as the multiple dispatch is concerned, the default-value methods are managed as normal methods. They are statically type checked as normal methods. But, since we wanted them attached to a unique instance variable, their invocation is odd: it occurs always in `assign` where the effective default-value method name is actually constructed right before of the invocation; the abstraction level for such invocation cannot be really computed and is stated as *Top*, just to allow the method lookup algorithm to run (there are only the constraints in Def. 11 to ensure that a "good" default-value is eventually computed).

4. The redefinition of multiply inherited instance variables leads to multiple default-value methods, that are managed straightforwardly by a class hierarchy transformation.

About the default-value methods technique we have to stress two aspects:

i) a default-value method is sharper than a plain "default value":

The default-value method allow to assign an instance variable by a value which is dependent on the context of the assignment in two ways: first, the way such value is computed is designed in the class in which it will act; second the value is computed on the basis of another value provided at run-time, so significant if not suitable for a direct assignment.

We think that we couldn't obtain the same result by means of the commonly allowed "default value", specified at class definition time. In this case we would have that for any troublesome assignments the instance variable assumes the same value, independently on the run-time context.

ii) default-value methods draw the edge between safety and unsafety:

As a matter of facts, this technique is not always applicable, i.e. there are cases in which we can't fix an instance variable redefinition because we can't reasonably design a suitable default-value method. This is just a way to confirm that using polymorphism with covariant redefinition of instance variables is not always safe: it must be supported for reasons of expressivity and flexibility, but its usage must be constrained.

The applicability of our technique can draw the edge between the cases of safety and unsafety.

In Fig. 11 we show a set of definitions for the last classes of Fig. 1: a course now specifies a *device* to be used during lessons, and an *assistant* that manages such a device. This assistant is redefined covariantly in *Special_Course*. The program statements present again an update problem. This time there is no default-value method to let the bad assignment `c.Set_dev_Assistant(a)` recover safely. And this is because no reasonable definition of a `dev_assistant-default-value(p:`*Assistant*`):`*Special_Assistant* could be devised in *Special_Course*.

```
class Course { }
aTeacher:Teacher;
Days:string;
dev:Device;
dev_assistant:Assistant;
Set_dev_assistant(p:Assistant)
is {dev_assistant := p };
endclass
```

```
class Special_Course { Course}
aTeacher:Special_Teacher;
dev:Special_Device;
dev_assistant:Special_Assistant;
Set_dev_assistant(p:Special_Assistant)
is {dev_assistant := p };
endclass
```

```
c:Course
a:Assistant
c := new Course
a := new Special_Assistant
c.Set_dev_assistant(a)
```
(normal assistant attached to special course)

Figure 11: a polymorphic school II

In other words, the constraint, that polymorphism is supported iff a suitable default-value method is defined for each instance variable redefinition, is what prevents the specialization bounded polymorphism from being unsafe at run-time.

# 5 Conclusions

In this paper we have rather informally discussed the problems arising in an object-oriented language when covariant redefinition of class attributes is supported. We have called *specialization inheritance* the enhanced strict inheritance where such a feature s supported. We have seen that in such a framework we cannot basically trust a program that uses polymorphic assignments and method invocations.

In Sec. 3 we saw that polymorphic assignments and method invocations can safely be supported, in a specialization inheritance framework, whenever there is no instance variable redefinition occurring. The covariant redefinition of methods in subclassing is made possible and safe, by exploiting a multiple dispatch mechanism based on abstraction level and method linearization. The characteristics of our proposal are in the use of the abstraction level to bound the method lookup, the use of method linearizations to ensure that there isn't a more specialized applicable method than the selected one, and the use of static method linearization for static type checking of method invocations. In particular we apply the restrictions of Def. 3 for the static correctness of method invocations.

In Sec. 4 we have shown a technique aimed to support also covariant instance

variable redefinition. If this technique is applicable, the programmer is requested of additional definitions (the default-value methods), but the program can safely execute polymorphic statements. Actually, this appears to be the maximal support that a programming system can give, being the covariant instance variable redefinition "the" reason for type unsafeness of the specialization inheritance (when it coexists with polymorphism).

A discussion of related work is provided in App. B. In particular App. B.2 discusses about covariant redefinition of methods and App. B.3 provides comparisons with two recent related approaches in our knowledge (one not yet published and available via file transfer protocol).

Our support to specialization inheritance is very flexible and allow for object-oriented programming very close to the development schema of object-oriented design. Our proposals related to method specialization result in a burden only for the language implementor (as we are experimenting), while the programmer is provided with the very natural mechanism of covariant redefinition. The proposals related to instance variable specialization are more demanding for the programmer, in terms of default-value methods definitions.

# References


[1] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. In A. Paepke, editor, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '91, Phoenix, Arizona, USA, Oct. 6–11, 1991*, pages 113–128. ACM Press, 1991. (Available as SIGPLAN NOTICES 26, 11, Nov. 1991).

[2] A. Albano, L. Cardelli, and R. Orsini. Galileo: a strongly typed interactive conceptual language. *ACM Trans. on DataBase Systems*, 10(2):230–260, June 1985.

[3] D. G. Bobrow, L. G. DeMichiel, P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification, X3JI3 doc. 88-002R. *Sigplan Notices*, 23(9), 1988. Special Issue.

[4] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: a practical case. In Pierre Cointe, editor, *Proc. European Conference on Object-Oriented Programming,ECOOP'96, Paris, France, Jul., 1996*, number 1098 in Lecture notes in computer science, pages 3–25, Berlin Heidelberg New York Tokio, 1996. Springer.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Comput. Surv.*, 17:471–522, 1986.

[6] B. Carré and J. M. Geib. The point of view notion for multiple inheritance. In N. Meyrowitz, editor, *Proc. Joint Conf. "ACM Conference on Object-Oriented Programming: Systems, Languages and Applications" and "European Conference on Object-Oriented Programming, Ottawa, Canada, Oct. 21–25, 1990"*, pages 312–321. ACM Press, 1990. (Available as SIGPLAN NOTICES, 25, 10, Oct. 1990).

[7] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Trans. on Programming Languages and Systems*, 17(3):431–447, 1995.

[8] G. Castagna. Instance variable specialization in object-oriented programming. submitted for publication, available at URL *http://www.dmi.ens.fr/∼castagna/pub.html*, 1996.

[9] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *Seventeenth ACM Conference on LISP and Functional Programming, San Francisco, CA, June 22–24, 1992*, pages 182–192, New York, July 1992. ACM Press.

[10] P. Di Blasio and M. Temperini. $\lambda\&_{ESI}$-calculus for enhanced strict inheritance. Technical Report R.401, Istituto di Analisi dei Sistemi ed Informatica, Consiglio Nazionale delle Ricerche, Italia, Feb. 1995. Available via *anonymous ftp* at *ftp.dis.uniroma1.it*, file *pub/marte/papers/calculus.ps.gz* or via URL *http://www.dis.uniroma1.it/pub/marte/homepage/publications.html*.

[11] P. Di Blasio and M. Temperini. Subtyping inheritance and its application in symbolic computation systems. *Jour. Symbolic Computation*, 19(1):39–63, 1995.

[12] P. Di Blasio, M. Temperini, and P. Terlizzi. Enhanced strict inherirance in TASSO-L. In A. Miola and M. Temperini, editors, *Advances in the Design of Symbolic Computation Systems*, Texts and monographs in symbolic computation, pages 179–195. Springer, Wien New York, 1997.

[13] R. Ducournau and M. Habib. On some algorithms for multiple inheritance in object-oriented programming. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Liebermann, editors, *Proc. European Conference on Object-Oriented Programming, ECOOP'87, Paris, France, June 1987*, volume 276 of *Lecture notes in computer science*, pages 243–252, Berlin, 1987. Springer.

[14] G. Ghelli. A static type system for message passing. In A. Paepke, editor, *Proc. ACM Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA '91, Phoenix, Arizona, USA, Oct. 6–11, 1991*, pages 129–145. ACM Press, 1991. (Available as SIGPLAN NOTICES 26, 11, Nov. 1991).

[15] A. Kreczmar, A. Salwicki, and M. Warpechowski. *LOGLAN '88 - Report on the Programming Language*, volume 414 of *Lecture notes in computer science*. Springer, Berlin Heidelberg New York Tokio, 1990.

[16] B. Meyer. *Eiffel: the Language*. Prentice Hall, 2nd edition, 1992.

[17] B. Meyer. Static typing and other mysteries of life. In *Proc. OOPSLA'94 and Proc. TOOLS PACIFIC 94*. ACM Press, 1994. Text of the keynote lecture, available at URL *http://www.eiffel.com/doc/manuals/technology/*.

[18] B. Meyer. Beware of polymorphic catcalls. Available at URL *http://www.eiffel.com/doc/manuals/technology/*, 1995.

[19] B. Stroustrup. *The C++ Programming Language*. Prentice Hall, 1991.

# A An algorithm for computing method linearizations

Define $\prec$ as ($<_{\mathcal{M}}$ *or* $\not>_{\mathcal{M}}$), and $\succ$ as $\not<$.

Here we provide an alternative sorting algorithm, by the following function *linearize*. Given a class $C$, a method `m(P):R` defined in it, the $C$`::m`'s method linearization is obtained as a sorting of the $C$ class linearization, calling *linearize*(`m`, $C$-linearization).

In the following, $\mathcal{C}$ stands for a list of classes.

The function *check*, given a method `m` and a list of classes, checks whether `m` precedes by $\prec$ all the homonymous methods defined in the classes of the list. In such case returns *nil*, otherwise the class whose method is strictly more specialized than `m`.

*linearize* (`m`, $\mathcal{C}$) =
  **let** $c$ = *check*(*car*($\mathcal{C}$)`::m`, *cdr*($\mathcal{C}$)) **in**
    **if** $c$ = *nil*
      **then return**(*cons*(*car*($\mathcal{C}$)), *linearize*(*cdr*(`m`, $\mathcal{C}$)))
      **else return** (*linearize* (*cons* ($c$, {$\mathcal{C} \setminus c$})))
**end** *linearize*

*check* (`m`,$\mathcal{C}$) =
  **if** *empty*($\mathcal{C}$)
    **then return** (*nil*)
    **else if** `m` $\prec$ *car*($\mathcal{C}$)
      **then return** (*check* (`m`, *cdr*($\mathcal{C}$)))
      **else return** (*car*($\mathcal{C}$))
**end** *check*

For example, the method linearization obtained through *linearize* for the hierarchy of Fig. 3 is {$D'$, $A$, $B_7$, $B_6$, $B_5$, $B_4$, $C$, $B_2$, $B_3$, $B_1$, $B$, $D$}.

An evidence of the termination of our algorithm can be seen by considering that the only source for never ending loops, in *linearize*, is the existence in a $C$-linearization {$C_1$, $C_2$, ..., $C_n$} of a cycle such as $C_{h_1}$`::m` $\succ$ $C_{h_2}$`::m` $\succ \cdots \succ$ $C_{h_k}$`::m` $\succ$ $C_{h_1}$`::m`. In this case we would have that after a *check* $C_{h_2}$`::m` became first for the next *linearize*; then $C_{h_3}$`::m` would become first, and so on, until $C_{h_1}$`::m` became first again.

We can exclude such occurrences, since $\succ$ is transitive and irreflexive. This statement follows by simple checkings on Tab. 3 ($P * P'$ means that anyone of the relations $C < C'$, $C > C'$, $C = C'$ can hold between the classes $P$ and $P'$):

**(irrefl.)** $\mathtt{m_i} \not\succ \mathtt{m_i}$.

If $\mathtt{m_i} \succ \mathtt{m_i}$ then either $\begin{cases} P_i > P_i \\ R_i = R_i \end{cases}$ or one of the cases $\begin{cases} P_i * P_i \\ R_i > R_i \end{cases}$ must hold.
And none is possible.

**(trans.)** $\mathtt{m_i} \succ \mathtt{m_k} \succ \mathtt{m_j} \Longrightarrow \mathtt{m_i} \succ \mathtt{m_j}$.

The following table shows the possible class relationships descending from the hypotheses:

| hyp. | relation that must hold | | |
|---|---|---|---|
| | either | or one out of | |
| $\mathtt{m_i} \succ \mathtt{m_k}$ | $\begin{cases} P_i > P_k \\ R_i = R_k \end{cases}$ | $\begin{cases} P_i * P_k \\ R_i > R_k \end{cases}$ | r1. |
| $\mathtt{m_k} \succ \mathtt{m_j}$ | $\begin{cases} P_k > P_j \\ R_k = R_j \end{cases}$ | $\begin{cases} P_k * P_j \\ R_k > R_j \end{cases}$ | r2. |
| | c1. | c2. | |

Then all the possible configurations derived from the hypotheses are listed, together with the consequence. A configuration is a couple $< \mathrm{r}i, \mathrm{c}j >, i, j \in (1, 2)$.

$$< \mathrm{r1}, \mathrm{c1} > \Rightarrow \begin{cases} P_i > P_j \\ R_i = R_j \end{cases} \qquad < \mathrm{r1}, \mathrm{c2} >, < \mathrm{r2}, \mathrm{c1} >, < \mathrm{r2}, \mathrm{c2} > \Rightarrow \begin{cases} P_i * P_j \\ R_i > R_j \end{cases}$$

and from all the consequences, $\mathtt{m_i} \succ \mathtt{m_j}$ follows.

# B Related work

## B.1 on the conflict resolution problem

In Fig. 12 an example of multiple inheritance with conflict is shown. By multiple inheritance, the *Doctor&Fellow* class must have all the behaviors of its superclasses. So we want it to inherit methods `trip-funding` from both *DoctorStudent* and *FellowResearcher*.

At both invocation `a[k]`$^{\texttt{Doctor\&Fellow}}$`.trip_funding(gtf)` (case **(1)**) and `b[h]`$^{\texttt{Doctor\&Fellow}}$`.trip_funding(gtf)` (case **(2)**) a conflict occurs.

---

class *ResearchPerson* {}
`trip_founding(t:`*`TripFolder`*`):`*`Money`* is {$\cdots$};
endclass

class *DoctorStudent* { *ResearchPerson*}
`trip_founding(t:`*`GrantedTripFolder`*`):`*`CashMoney`* is {$\cdots$};
endclass

class *FellowResearcher* { *ResearchPerson*}
`trip_founding(t:`*`GrantedTripFolder`*`):`*`Check`* is {$\cdots$};
endclass

class *Doctor&Fellow* { *DoctorStudent, FellowResearcher* }
endclass

note: *CashMoney,Check* $leq_{ESI}$ *Money*
*GrantedTripFolder* $\leq_{ESI}$ *TripFolder*

```
a:array[1..N]of DoctorStudent;
b:array[1..M]of FellowResearcher;
df:Doctor&Fellow;
gtf:GrantedTripFolder;
k, h: indices
...
a[k]:= new Doctor&Fellow;
···a[k].trip_funding(gtf)···;          (1)
...
b[h]:= new Doctor&Fellow;
···b[h].trip_funding(gtf)···;          (2)
```

Figure 12: conflict resolution problem in multiple specialization inheritance

---

There are several ways to solve this problem: one is to redefine the conflicting methods ([9, 7]) but this can be quite cumbersome, because programmers have to re-implement most of the methods in the subclass, even if it is not strictly

necessary. Another solution is to rename some conflicting methods ([16]). A third solution (still implying some user interaction to modify the program or the class hierarchy), consists in selecting the method to be executed right in the body of the invocation. The troubles arising from this solution (it means to manage an *arbitrary local precedence*, cf. [1]) are discussed in [6].

Other solutions are based on the use of multiple dispatch over class linearizations. In these approaches the whole cone of superclasses of the receiving object is visited, and this makes type errors possible. In particular, since the executable method could be selected from a class unrelated to the abstraction level, a result incompatible with the context of the method invocation could be returned. To avoid this troubles, the static type checking must ensure the *result type compatibility* among all the *confusable methods* ([1, 14]). For example, the methods $A$::m, $B_4$::m, $B_5$::m, $B_7$::m, $C$::m in Fig. 3 should be related such that the five result classes are ordered in a chain. This turns out again in user interventions, and in difficulties in software reuse. This also limits the flexibility of a language and narrows the use of polymorphism to a great extent.

Our *Enhanced Strict Inheritance* imposes no constraints other than the covariance of the redefinitions. For our specialization inheritance we adopt a different conflict resolution strategy, based on the use of abstraction level. Once we have given an upper bound to the method lookup, as described previously, we can ensure that the class of the object returned by a method invocation is correct w.r.t. the expected one, no matter whether there are confusable methods with incompatible result (and argument) classes.

Note that, in Fig. 12, results are not compatible: no redefinition of `trip_funding` in *Doctor&Fellow* makes sense, since this method should be redefined just choosing one of the two methods in the superclasses; renaming is possible, but would forbid polymorphic use of *Doctor&Fellow* objects.

Actually, an object of class *Doctor&Fellow* has different abstraction levels, depending on the context of the method invocations it receives: in case **(1)** it is a particular *DoctorStudent* (the static class / abstraction level of the variable `a[k]` that refers to it); in case **(2)**, a particular *FellowResearcher*.

In case **(1)**, *DoctorStudent*::`trip_funding` makes the invocation statically correct. When it is executed, `a[k]` has run-time class *Doctor&Fellow*, so the executable method is looked up starting from the lower bound *Doctor&Fellow*, where it is not found. The lookup proceeds towards the abstraction level, so that *DoctorStudent*::`trip_funding` is found applicable. A symmetric behavior is followed for the method invocation of case **(2)** (*FellowResearcher*::`trip_funding` makes the invocation statically correct, and *FellowResearcher*::`trip_funding` is found applicable).

So, our method lookup strategy ensures that no result type problem occurs, also in presence of confusable methods. Sometimes, by avoiding that some confusable methods join the search space, we get also freedom from conflict resolution problems at all, like in the above example. Of course this happens only in se-

lected cases. The general case involves occurrences of multiple inheritance that are not "cut off" by the search space. Turning back to the example of Fig. 12, under the declaration `rp:` *Research_Person*, the abstraction level of an object referred to by `rp` in `rp.trip_funding(···)` is *Research_Person*. So both *Doctor_Student*::`trip_funding` and *Fellow_Researcher*::`trip_funding` are executable. In these cases, if none of the constraints we mentioned above is adopted, when there are (or might be) several "equally applicable" methods, the most frequent attitude is to resort to the search order, and select the first method that met applicable. We have tried to contrast this semantic unclearness by the method linearization technique later in Sec. 3.

### B.2 on covariant redefinition of methods

The main achievement in our specialization inheritance is that we impose no constraints in programming on a class hierarchy, above the covariance of the redefinitions. The characteristics of our proposal are in

1. The use of the abstraction level to bound the method lookup.
2. The use of method linearizations to ensure that there isn't a more specialized applicable method than the selected one.
3. The use of static method linearization for static type checking of method invocations. In particular we apply the restrictions of Def. 3 for the static correctness of method invocations.

In fact, the idea we present is new w.r.t. similar approaches in literature. As far as we know of such approaches, the first trace of a statically bounded method lookup can be found in [1]. The original formulation of the abstraction level bounded method lookup is presented in [11] and developed in [10]. A quite similar approach has been independently presented in [4].

All these approaches are too permissive in the determination of the upper vertex.

In [1] the static type checking of multimethods is studied in the usual functional environment of multimethods. The functions contained in a generic function are studied and partitioned into a graph to allow for their selection. *Confusable methods* are topologically ordered. What cannot be ordered is collected in *blobs*, that wrap the possible conflict problem. A subset of the confusable set is saved for run-time dispatch. Blobs can possibly be singleton elements of such subset and need a run-time processing to solve the arisen conflict. In this saving of significant subgraph of the inheritance hierarchy there is a similarity with the diamond technique we propose. If we had to interpret (or to force) the [1] approach in our framework we would see that the class that makes the invocation statically correct can be any superclass of the receiver static type, and the "diamond" is the full set of subclasses of the upper vertex and superclasses of the receiver's actual class. This reflects what is done usually in multimethod languages, but so the diamond is too big and

it can contain classes that are unrelated to the receiver's static class. So there is the constraint, to be maintained by the programmer, that all the confusable methods have a compatible result type.

The comparison between the [1] approach and ours is in Fig. 13. In part **(a)** we suppose that an invocation such as $\mathtt{a}^D\mathtt{.m}(\cdots)$ is to be executed. Assume $S$ is the static class of `a` and that $O\mathtt{::m}$ is statically applicable. Assume also that `m` is not explicitly defined neither in $S$ nor in any class in $\mathcal{S}^{S,O}$ (but $O$ itself). Then both [1] and us would declare the invocation correct: we with abstraction level $O$, [1] saving the subgraph $\mathcal{S}^{D,O}$ for the run-time dispatch. The only difference appears to be that our search space is limited to $\mathcal{S}^{D,S} \cup \mathcal{S}^{S,O}$, instead of the whole $\mathcal{S}^{D,O}$. A second difference is in that we accept less method invocations, due to the restriction made by Def. 3. In Fig. 13, part **(b)**, maintain the previous assumptions, but suppose there is a (re)definition $A\mathtt{::m}$ in $\mathcal{S}^{S,O}$, and that this definition is not statically applicable: then we wouldn't accept the invocation. It would be correct in [1] with the same subset $\mathcal{S}^{D,O}$ as search space at run-time.

Our present approach derives from [11, 10], where the two-diamond technique was introduced, just with too big upper diamond. In part part **(b)** of Fig. 13, the previous method invocation would be statically correct and the indicated double diamond would be the search space. As in [1], the method selection is made along a linearization and the first method that is found applicable is selected. Def. 3 is absent and the conflict resolution problem is solved with no guarantee that another method would be more specialized than the selected.

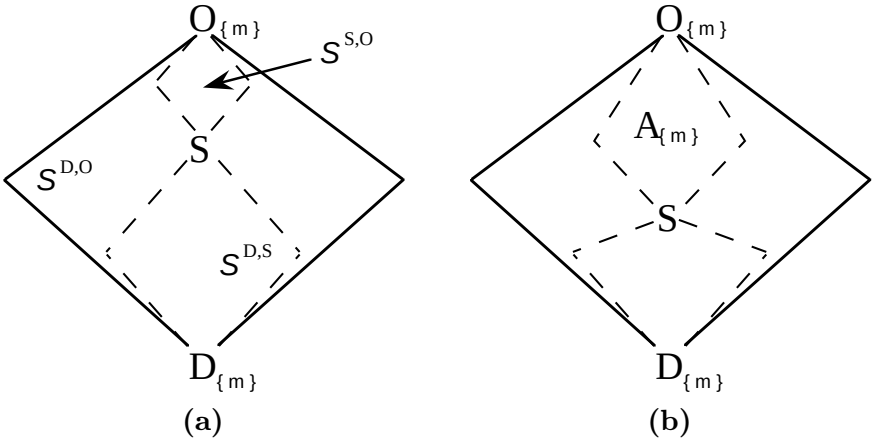



Figure 13: different approaches to the abstraction level

In [4] the problem of covariant method redefinition is considered in the framework of the $\lambda\&$ calculus of overloaded functions. Methods are defined as sets of functions (*branches*) in a multimethod. The main point is that if one wants to

override a method by a redefinition in a subclass, s/he has also to add an additional branch to handle the arguments that could be passed to the overridden method but not to the overriding one. Finally this addition is made automatic: if the redefinition isn't applicable, then the previous method is called. (In our terms we could say that the method lookup proceeds by being pushed from a subclass to a superclass.) For each new redefinition, new branches are added automatically to manage by calling older methods where the new method cannot work.

Interpreting this approach in our framework shows that the "diamond" could be thought as the whole $\mathcal{S}^{D,O}$ in Fig. 13, part **(a)**. Actually it is the static class $S$ that makes the invocation correct, but at run-time the multiple dispatch can be "pushed" over, until an applicable method (branch) is found. Since we know that at most in $O$ there is an applicable method we could think that the search space is at most $\mathcal{S}^{D,O}$.

Another difference is in the treatment of conflicts: if there are several superclasses where to be pushed to look for an applicable method, then the method lookup jumps over them, and climbs till the first common superclass.

There is no result compatibility problem and always the really most specialized method is selected, since further constraints are imposed on the programmer: in particular for each method invocations there must be a *least* function right to answer, so the programmer has to redefine a method each time it is multiply inherited.

As stated previously (Sec. 3.3), we try to work with our inheritance without the constraints adopted by other approaches. Here we allow to check whether a class makes an invocation statically correct by checking only its methods that are either explicitly (re)defined or directly inherited. In this way we avoid that expressions are charged at compile-time with duties (answer a method invocation: perform a behavior) that are too general for them already at compile-time. If an expression yields an object of static class $T$, then we want that its behavior is checked at compile-time only on the basis of the $T$ static features. On the other hand, if at run-time the expression yields an object of class $T' \leq T$, then we can use the behavioral heritage received from $T$ (since it was tested in a $T$ environment).

Any other approach (to our knowledge) admits to use higher classes than the static class, to help declaring a method invocation statically correct.

Finally, all the semantic unclearness due to a multiple dispatch strategy based on linearization is wiped off by the choice of method linearizations in place of the fixed class linearization. Any other approach (to our knowledge) either disregards the possibility of selecting a method while another is applicable and more specialized, or imposes that a least method is defined among the confusable ones (by forced redefinition).

### B.3 on covariant redefinition of instance variables

In current object-oriented programming languages where covariant instance variable redefinition is supported, the use of polymorphism is left unsafe (as in Loglan [15])

or forbidden (may be after *system level type checking* [17, 18]). The latter appears to be a too strong solution, preventing a great deal of programs from being executed. On the other hand this is the only solution if one wants to just get old programs in the new, safer environments.

W.r.t. this solution, ours appear to be more expressive, since we can accept much more programs still safe.

A (very!) recent approach is given in the submitted paper [8], featuring a solution extremely similar to the one presented here. It is developed on the ground of the $\lambda\&$ calculus ([9, 7]) and manages the definition of *conversion functions* acting as ours default-value methods. The $\lambda\&$ calculus is enhanced by imperative features and *specializable locations*, obtaining the $\lambda\&^{:=}$ calculus. A specializable location is a couple [M,N] where M is a location reference and N is a conversion function. If an assignment is done on M, the value to be assigned is sieve through the conversion function, so to get a value of the right type by which to assign. The conversion function is actually an overloaded function containing the local conversion functions and their composition with the previously defined conversions and all the needed compositions among previously defined conversions. All these compositions are made automatically. The definition of additional conversion functions is mandatory, for the cases of occurring conflict resolution problems in building the compositions to put in N.

W.r.t. this solution, ours doesn't provide formal subtyping and solves the conflict problem without obliging to define additional default-value methods.